Legal Information

DirectX 5
Programmer's Reference

Introducing DirectX 5

DirectX® 5 provides strategies, technologies and tools that can help you build the next generation of computer games and multimedia applications. This overview covers general introductory information about the DirectX 5 Programmer's Reference in the Platform Software Development Kit (SDK) documentation. Information is divided into the following sections:

DirectX Goals

The Microsoft® DirectX® Programmer's Reference provides a finely tuned set of application programming interfaces (APIs) that provide you with the resources you need to design high-performance, real-time applications. DirectX technology will help build the next generation of computer games and multimedia applications.

Microsoft developed DirectX so that the performance of applications running in the Microsoft Windows® operating system can rival or exceed the performance of applications running in the MS-DOS® operating system or on game consoles. This Programmer's Reference was developed to promote game development for Windows by providing you with a robust, standardized, and well-documented operating environment for which to write games.

DirectX provides you with two important benefits:

- Benefits of Developing DirectX Windows Applications
- Providing Guidelines for Hardware Development

Benefits of Developing DirectX Windows Applications

When Microsoft created DirectX, one of its primary goals was to promote games development for the Windows environment. Prior to DirectX, the majority of games developed for the personal computer were MS-DOS-based. Developers of these games had to conform to a number of hardware implementations for a variety of cards. With DirectX, games developers get the benefits of device independence without losing the benefits of direct access to the hardware. The primary goals of DirectX are to provide portable access to the features used with MS-DOS today, to meet or improve on the performance of MS-DOS console-based applications, and to remove the obstacles to hardware innovation on the personal computer.

Additionally, Microsoft developed DirectX to provide Windows-based applications with high-performance, real-time access to available hardware on current and future computer systems. DirectX provides a consistent interface between hardware and applications, reducing the complexity of installation and configuration and using the hardware to its best advantage. By using the interfaces provided by DirectX, software developers can take advantage of hardware features without being concerned about the implementation details of that hardware.

A high-performance Windows-based game will take advantage of the following technologies:

- Accelerator cards designed specifically for improving performance
- Plug and Play and other Windows hardware and software
- Communications services built into Windows, including DirectPlay

Providing Guidelines for Hardware Development

DirectX provides hardware development guidelines based on feedback from developers of high-performance applications and independent hardware vendors (IHVs). As a result, the DirectX Programmer's Reference components might provide specifications for hardware-accelerator features that do not yet exist. In many cases, the software emulates these features. In other cases, the software polls the hardware regarding its capabilities and bypasses the feature if it is not supported.

The DirectX Programmer's Reference

This section describes the DirectX Programmer's Reference components and some DirectX implementation details. The following topics are discussed:

DirectX Programmer's Reference Components

The DirectX Programmer's Reference includes several components that address the performance issues of programming Windows-based games and high-performance applications. This section lists these components and provides a link for more information on each component.

- DirectDraw® 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. For more information, see About DirectDraw in the DirectDraw documentation.
- DirectSound® enables hardware and software sound mixing and playback. For more information, see About DirectSound in the DirectSound documentation.
- DirectPlay® makes connecting games over a modem link or network easy. For more information, see About DirectPlay in the DirectPlay documentation.
- Direct3D® provides a high-level Retained-Mode interface that allows applications to easily implement a complete 3-D graphical system, and a low-level Immediate-Mode interface that lets applications take complete control over the rendering pipeline. For more information about Immediate Mode, see About Direct3D Immediate Mode. For more information about Retained Mode, see About Retained Mode.
- DirectInput® provides input capabilities to your game that are scalable to future Windows-based hardware-input APIs and drivers. Currently the joystick, mouse, keyboard, and force feedback devices are supported. For more information, see About DirectInput in the DirectInput documentation.
- DirectSetup provides a one-call installation procedure for DirectX. For more information, see About DirectSetup in the DirectSetup documentation.
- AutoPlay is a Windows 95 feature that starts an installation program or game automatically from a compact disc when you insert the disc in the CD-ROM drive. For more information, see About AutoPlay in the AutoPlay documentation.

The AutoPlay feature is part of the Microsoft Win32® API in the Platform SDK and is not unique to DirectX.

Among the most important parts of the documentation for the DirectX Programmer's Reference is the sample code. Studying code from working samples is one of the best ways to understand DirectX. Sample applications are located in the Sdk\Samples folder of the Platform SDK or in the DirectX code samples under the Platform SDK References section.

Detecting DirectX Versions

You can determine which version of DirectX is installed on a system by querying for various DirectX object interfaces. The **GetDXVersion** sample function from the GetDXVersion.cpp file in the \SDK\Samples\Misc directory performs this task for you. The function creates a few key DirectX objects − a DirectDraw object, a DirectDrawSurface object, and a DirectInput object − then queries for interfaces that were introduced in previous releases of DirectX. The function determines which version of DirectX is installed on a system and the installed operating system by using a simple process of elimination.

If you have previous versions of the DirectX SDK on your system, make sure you do not have those files in your include path or library (.LIB) path. Inadvertently linking with DirectX 3 libraries after compiling with DirectX 5 header files will cause your application to behave unpredictably.

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.

The following example uses the **IDirectDraw2_CreateSurface** macro to call the **IDirectDraw2::CreateSurface** method. The first parameter is a reference to the DirectDraw object that has been created and invokes the method:

```
ret = IDirectDraw2_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.

DirectX and the Component Object Model

This section describes the Component Object Model (COM) and how it implements the DirectX objects and interfaces. The following topics are discussed:

The Component Object Model

Most APIs in the DirectX Programmer's Reference are composed of objects and interfaces based on the COM. The COM is a foundation for an object-based system that focuses on reuse of interfaces, and it is the model at the heart of COM 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 APIs are created as instances of COM *objects*. You can consider an object to be 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 **IDirectDraw2::GetDisplayMode** method is sent through the **IDirectDraw2** 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 they can use the implementation of interfaces provided by the other object. If you know an object is an COM object, and if you know which interfaces that object supports, your application (or another object) can determine which services the first object can perform. One of the methods all COM objects inherit, the **QueryInterface** method, lets you determine which interfaces an object supports and creates pointers to these interfaces. For more information about this method, see the IUnknown Interface.

IUnknown Interface

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

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

The **AddRef** and **Release** methods maintain an object's reference count. For example, if you create a DirectDrawSurface object, the object's reference count is set to 1. Every time a function returns a pointer to an interface for that object, the function then must call **AddRef** through that pointer to increment the reference count. You must match each **AddRef** call with a call to **Release**. Before the pointer can be destroyed, you must call **Release** through that pointer. After an object's reference count reaches 0, the object is destroyed and all interfaces to it become invalid.

The **QueryInterface** method determines whether an object supports a specific interface. If an object supports an interface, **QueryInterface** returns a pointer to that interface. You then can 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.

IUnknown::AddRef

The **IUnknown::AddRef** method increases the reference count of the object by 1.

ULONG AddRef();

Parameters

None.

Return Values

Returns the new reference count.

Remarks

When the 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 **Release** method to decrease the object's reference count by 1.

This method is part of the **IUnknown** interface inherited by the object.

IUnknown::QueryInterface

The **IUnknown::QueryInterface** method determines if the object supports a particular COM interface. If it does, the system increases the object's reference count, and the application can use that interface immediately.

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

Parameters

riid

Reference identifier of the interface being requested.

obp

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

Return Values

If the method succeeds, the return value is S_OK.

If the method fails, the return value is E_NOINTERFACE or one of the following interface-specific error values. Interface-specific error values are listed by component.

DirectDraw

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY (**IDirectDrawSurface2** and **IDirectDrawSurface3** only)

DirectSound

DSERR_GENERIC (**IDirectSound** and **IDirectSoundBuffer** only)

DSERR_INVALIDPARAM

DirectPlay

DPERR_INVALIDOBJECT

DPERR_INVALIDPARAMS

For Direct3D Retained-Mode and Immediate-Mode interfaces, the **QueryInterface** method returns one of the values in Direct3D Retained-Mode Return Values and Direct3D Immediate-Mode Return Values.

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 Microsoft and third parties to extend objects without interfering with each other's existing or future functionality.

This method is part of the **IUnknown** interface inherited by the object.

IUnknown::Release

The **IUnknown::Release** method decreases the reference count of the object by 1.

ULONG Release();

Parameters

None.

Return Values

Returns the new reference count.

Remarks

The object deallocates itself when its reference count reaches 0. Use the **AddRef** method to increase the object's reference count by 1.

This method is part of the **IUnknown** interface inherited by the object.

DirectX COM Interfaces

The interfaces in the DirectX Programmer's Reference have been created at a very basic level of the COM programming hierarchy. Each interface to an object that represents a device, such as **IDirectDraw2**, **IDirectSound**, and **IDirectPlay**, derives directly from the **IUnknown** COM interface. Creation of these basic objects is handled by specialized functions in the dynamic-link library (DLL) for each object, rather than by the **CoCreateInstance** function typically used to create COM objects.

Typically, the DirectX object model provides one main object for each device. Other support service objects are derived from this main object. For example, the DirectDraw object represents the display adapter. You can use it to create DirectDrawSurface objects that represent the display memory 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 use 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 that the interface exists on an object and to obtain a pointer to that interface. After sending **QueryInterface**, your application or object actually receives from the object a pointer to the vtable, through which this method can call the interface methods implemented by the object. This mechanism isolates from one another any private data the object uses and the calling client process.

Another similarity between COM objects and C++ objects is that a method's first argument is the name of the interface or class, called the *this* argument in C++. Because COM objects and C++ objects are completely 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++.

Retrieving Newer Interfaces

The component object model dictates that objects update their functionality not by changing the methods within existing interfaces, but by extending new interfaces that encompass new features. In keeping existing interfaces static, an object built on COM can freely extend its services while maintaining compatibility with older applications.

DirectX components following this philosophy. For example, the DirectDraw component supports three versions of the **IDirectDrawSurface** interface: **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3**. Each version of the interface supports the methods provided by its ancestor, adding new methods to support new features. If your application doesn't need to use these new features, it doesn't need to retrieve newer interfaces. However, to take advantage of features provided by a new interface, you must call the object's **IUnknown::QueryInterface** method, specifying the globally unique identifier (GUID) of the interface you want to retrieve. Interface GUIDs are declared in the corresponding header file.

The following example shows how to query for a new interface:

```
LPDIRECTDRAW    lpDD1;
LPDIRECTDRAW2   lpDD2;

ddrval = DirectDrawCreate( NULL, &lpDD1, NULL );
if( FAILED(ddrval))
    goto ERROROUT;

// Query for the IDirectDraw2 interface
ddrval = lpDD1->QueryInterface(IID_IDirectDraw2, (void **)&lpDD2);
if( FAILED(ddrval))
    goto ERROROUT;

// Now that we have an IDirectDraw2, release the original interface.
lpDD1->Release();
```

In some rare cases, a new interface will not support some methods provided in a previous interface version. The **IDirect3DDevice2** interface is an example of this type of interface. If your application requires features provided by an earlier version of an interface, you can query for the earlier version in the same way as shown in the preceding example, using the GUID of the older interface to retrieve it.

Accessing COM Objects by Using C

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

- The first parameter of the method always refers to the object that has been created and that invokes 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 **IDirectDraw2::CreateSurface** method with the C programming language:

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

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

To call the **IDirectDraw2::CreateSurface** method, first dereference the DirectDraw object's vtable, and 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 which invokes 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 *lpVtbl* parameter and passes the *this* pointer):

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

Interface Method Names and Syntax

All 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 you can use these methods only 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. The following example shows the C++ syntax for the **IDirectDraw2::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.

What's New in the DirectX 5 Programmer's Reference?

The DirectX 5 Programmer's Reference provides more services—and more avenues for innovation—than did the DirectX 3 documentation. (Note that there is no "DirectX 4"—the numbering jumps from version 3 to version 5.) Although this Programmer's Reference contains additional functions and services, all the applications you wrote with previous DirectX APIs will compile and run successfully without changes.

The purpose of this section is to help those of you who are familiar with DirectX 3 quickly identify several important areas of this Programmer's Reference that are significantly different. These differences are listed by component.

DirectDraw

DirectDraw has been extended with new video-port capabilities that allow applications to control the flow of data from a hardware video-port device to a DirectDraw surface in display memory. For an overview of the video-port extensions, see Video Ports.

Additionally, the DirectDraw HEL now exploits performance improvements made possible by the Pentium MMX processor. DirectDraw tests for the presence of an MMX processor the first time you create a surface in any process. On non-Pentium machines, this test can cause a benign first-chance exception ("Illegal Instruction") to be reported by the debugger. The exception will not affect your application's performance or stability.

DirectDraw now supports off-screen surfaces wider than the primary surface. You can create surfaces as wide as you need, permitting that the display hardware can support it.

For more information, see Creating Wide Surfaces.

DirectDraw now supports the Advanced Graphics Port (AGP) architecture. On AGP-equipped systems, you can create surfaces in non-local video memory. The **DDSCAPS** structure now supports flags to differentiate between standard (local) video memory and AGP (non-local) video memory. The **DDCAPS** structure now contains members that carry information about blit operations using non-local video memory surfaces. For more information, see Using Non-local Video Memory Surfaces.

DirectSound

DirectSound includes a new interface, **IKsPropertySet**, that enables it to support extended services offered by sound cards and their associated drivers. For more information, see DirectSound Property Sets.

Also new is **DirectSoundCapture**, a COM-based wrapper for the Win32 **waveIn** functions that will be extended in the future to work directly with the drivers.

DirectPlay

DirectPlay includes a new interface, **IDirectPlay3**, that is exactly the same as **IDirectPlay2** with new methods. Similarly, **IDirectPlayLobby2** is an extended version of **IDirectPlayLobby**.

New functionality in DirectPlay includes the ability for applications to suppress service provider dialogs by creating connection shortcuts, asynchronous **EnumSessions** to keep an up-to-date list of available sessions, implementation of the **SetSessionDesc** method, better

support for password protected sessions, support for secure server connections and the ability to create multiple DirectPlay objects and to create them directly using **CoCreateInstance**.

For more information about these new features, see What's New in DirectPlay?

Direct3D

Direct3D Immediate Mode now supports drawing primitives without having to work directly with execute buffers. For more information, see The DrawPrimitive Methods. A set of extensions and helper functions has been implemented for C++ programmers; for more information, see D3D_OVERLOADS.

Direct3D Retained Mode now support interpolators that enable you to blend colors, move objects smoothly between positions, morph meshes, and perform many other transformations. Retained Mode also supports progressive meshes that allow you to begin with a coarse mesh and increasingly refine it; this can help you take the level of detail into account and can help with progressive downloads from remote locations. For more information, see the IDirect3DRMInterpolator Interface and the IDirect3DRMProgressiveMesh Interface

The Direct3D documentation has been updated for DirectX 5. The overview of Immediate Mode is more comprehensive, there is an Immediate-Mode tutorial, and there is a description of the .X file format.

DirectInput

DirectInput now provides COM interfaces for joysticks (a term that includes other input devices such as game pads and flight yokes) and for force feedback devices as well as for the mouse and keyboard.

The DirectInput documentation has been expanded to include reference material for the new functionality as well as new overviews and tutorials.

DirectSetup

DirectSetup now includes greater user interface customization capabilities. This is provided through a callback function that is passed to DirectSetup before it begins installing DirectX components and drivers. The callback function communicates the current installation status to your application's setup program. You can use this information to display the status through a user interface that is customized for your program.

In addition, DirectSetup now provides a way for multiplayer games that use DirectPlayLobby to remove their registration information from the registry.

AutoPlay

No changes for DirectX 5.

Conventions

The following conventions 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.
Bold text	Denotes a function, structure, macro, interface, method, data type, or other keyword in the programming interface, C, or C++.
[]	Encloses 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, the following 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 also are bold, and constants.
monospace	Sets off code examples and shows syntax spacing.

DirectDraw

This section provides information about the DirectDraw component. Information is divided into the following groups:

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

About DirectDraw

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

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

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

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

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

Why Use DirectDraw?

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

- The Hardware Abstraction Layer (HAL) of DirectDraw provides a consistent interface through which to work directly with the display and video memory, getting maximum performance from the system hardware.
- DirectDraw assesses the video hardware's capabilities, making use of special hardware features whenever possible. For example, if your video card supports hardware blitting, DirectDraw delegates blits to the video card, greatly increasing performance. Additionally, DirectDraw provides a Hardware Emulation Layer (HEL) to support features when the hardware does not.
- DirectDraw exists over Windows 95, gaining the advantage of 32-bit memory addressing and a flat memory model that the operating system provides. DirectDraw presents video and system memory as large blocks of storage, not as small segments. If you've ever used segment:offset addressing, you will quickly begin to appreciate this "flat" memory model.
- DirectDraw makes it easy for you to implement page flipping with multiple back buffers in full-screen applications. For more information, see Page Flipping and Back Buffering.
- Support for clipping in windowed or full-screen applications.
- Support for 3-D z-buffers.
- Support for hardware-assisted overlays with z-ordering.
- Access to image-stretching hardware.
- Simultaneous access to standard and enhanced display-device memory areas.
- Other features include custom and dynamic palettes, exclusive hardware access, and resolution switching.

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

Getting Started-Basic Graphics Concepts

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

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

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

Device-Independent Bitmaps

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

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

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

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

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

    if (hbm == NULL)
        return NULL;

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

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

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

```
        return NULL;

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

    DeleteObject(hbm);

    return pdds;
}
```

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

Drawing Surfaces

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

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

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

For more detailed information, see Surfaces.

Blitting Concepts

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

You can use the **IDirectDrawSurface3::Blt** and **IDirectDrawSurface3::BltFast** methods to perform blitting.

Page Flipping and Back Buffering

Page flipping is key in multimedia, animation, and game software. Software page flipping is analogous to the way cartoon artists animate their images. For example, the artist draws a figure on a sheet of paper, then sets it aside to work on the next frame. With each frame, the artist changes the figure slightly, so that when you flip between sheets rapidly the figure appears animated.

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

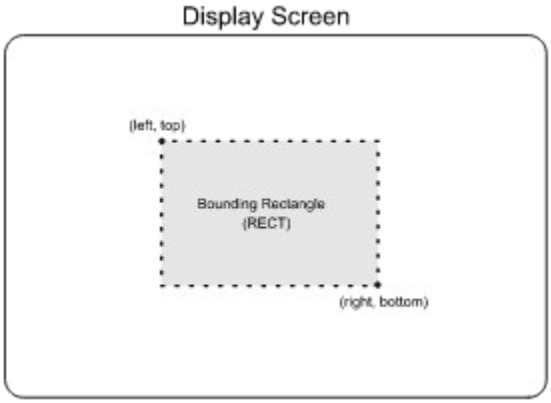
DirectDraw makes it easy for you to set up page flipping schemes, from a relatively simple double-buffered scheme (a primary surface with one back buffer) to more sophisticated schemes that add additional back buffers. For more information see DirectDraw Tutorials and Flipping Surfaces.

Introduction to Rectangles

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

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

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



Sprite Concepts

This section contains information about the basic concepts behind a common type of sprite animation. The following topics are discussed:

What is a Sprite?

Many video games use sprites. In its most basic sense, a sprite is an image that moves around on the screen. The sprite is drawn onto a surface over the existing background, the composed scene is sent to the screen, then the sprite is redrawn in a new location, and the process repeats. Combine this with repairing the sprite's old location on the background and page flipping and you get the illusion that a sprite is moving around the screen.

Transparent Blitting and Color Keys

One challenge in sprite animation is accommodating nonrectangular sprites, which means almost all of them. Because blitting functions work with rectangles (for efficiency, consistency, and ease of use), your sprites must fit into rectangles as well, whether or not they actually look rectangular on the screen.

Although the concept might be confusing at first, this is how it works: The sprite image itself is nonrectangular, but is contained in a rectangular space where every pixel that is not part of the sprite is treated as "transparent" when the blitter is moving the image to its destination. The artist creating the sprite chooses an arbitrary color that will be used as the transparency "color key." This is typically a single uncommon color that the artist doesn't use for anything but transparency, but it can also be a specified range of colors.

Using the **IDirectDrawSurface3::SetColorKey** method, you can set the color key for a surface. After the color key is set, subsequent **IDirectDrawSurface3::BltFast** method calls can take advantage of that color key, ignoring the pixels that match it. This type of color key is known as a source color key. Because the source color key prevents “transparent” pixels from being written to the destination, the original background pixels are preserved in these places, making it look like the sprite is non-rectangular object and passing over the background.

Additionally, you can use a color key that affects the destination surface (a destination color key). A destination color key is a color on a surface that is used for pixels that can be overwritten by a sprite. In this case, for example, the artist might be working on a foreground image that sprites are supposed to pass behind, creating a layered effect. Again, the artist chooses an arbitrary color that isn't used elsewhere in the image, reserving it as a portion of the image where you are allowed to blit. When you blit a sprite to the destination surface with a destination color key specified, the sprite's pixels will only be blitted to pixels on the destination that are using the destination color key. Because the normal destination pixels are preserved, it looks like the sprite passes behind the image on the destination surface.

Sprite and Patch Rectangles

To complete the illusion of sprite movement, you need a way to erase the sprite's image from the background before you draw it at its new location. You could reload the entire background and redraw the sprite, but a great deal of performance would be lost. Instead, you can keep track of the rectangle that is the sprite's last location and redraw only that portion. This method is called "patching." To patch the sprite's old location, redraw the sprite's old location with a copy of the original background image, which you previously loaded on an off-screen surface. The process works well, because it doesn't waste a lot of processing time blitting an entire surface each cycle.

This process can be described in the following simple steps:

1. Set the patch rectangle to the last sprite location.
2. Patch the background at that location by blitting to the background image from the off-screen master copy.
3. Update the sprite's destination rectangle to reflect its new location.
4. Blit the sprite to its newly updated rectangle in the background image.
5. Repeat.

Using straightforward C/C++ combined with the graphics power provided by DirectDraw, you can implement this process to make a simple sprite engine.

Bounds Checking and Hit Detection

Bounds checking and hit detection are two very common and important tasks associated with sprites. Bounds checking is a term used to describe the process of limiting a sprite's possible range of motion. For example, you might want to limit a given sprite to keep it from moving off the screen. To do so, you can check the values for the sprite's location, which you'll probably keep in a **RECT** structure, and prevent them from changing beyond the limits of the screen resolution. DirectDraw doesn't provide bounds checking services, but you can easily implement a bounds checking scheme in C/C++ alone. Clipping, on the other hand, is supported by DirectDraw. For more information, see Clippers.

Hit detection, or collision detection, refers to the process of checking whether one or more sprites occupy the same place. Most hit detection schemes involve checking to see if the bounding rectangles for one or more sprites overlap. Because there are so many different types of hit detection schemes with an equally varied number of uses, DirectDraw doesn't support them for you, thereby giving you the freedom to implement a hit detection scheme that meets your application's needs.

DirectDraw Architecture

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

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

Architectural Overview

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

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

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

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

The DirectDraw component provides services through COM-based interfaces. In the most recent iteration, these interfaces are **IDirectDraw2**, **IDirectDrawSurface3**, **IDirectDrawPalette**, **IDirectDrawClipper**, and **IDirectDrawVideoPort**. Note that, in addition to these interfaces, DirectDraw continues to support all previous versions. For more information about COM concepts that you should understand to create applications with the DirectX APIs in the Platform SDK, see DirectX and the Component Object Model .

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

After creating a DirectDraw object, you can create surfaces for it by calling the **IDirectDraw2::CreateSurface** method. Surfaces represent the memory on the display hardware, but can exist on either video memory or system memory. DirectDraw extends support for palettes, clipping (useful for windowed applications), and video ports through its other interfaces.

DirectDraw Object Types

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

The DirectDraw component uses the following objects:

DirectDraw object
The DirectDraw object is the heart of all DirectDraw applications. It's the first object you create, and you use it to make all other related objects. You create a DirectDraw object by calling the **DirectDrawCreate** function. DirectDraw objects expose their functionality through the **IDirectDraw** and **IDirectDraw2** interfaces. For more information, see The DirectDraw Object.

DirectDrawSurface object
The DirectDrawSurface object (casually referred to as a "surface") represents an area in memory that holds data to be displayed on the monitor as images or moved to other surfaces. You can create a surface by calling the **IDirectDraw2::CreateSurface** method of the DirectDraw object with which it will be associated. DirectDrawSurface objects expose their functionality through the **IDirectDrawSurface**, **IDirectDrawSurface2**, and **IDirectDrawSurface3** interfaces. For more information, see Surfaces.

DirectDrawPalette object
The DirectDrawPalette object (casually referred to as a "palette") represents a 16- or 256-color indexed palette to be used with a surface. It contains a series of indexed RGB triplets that describe colors associated with values within a surface. You do not use palettes with surfaces that use a pixel format depth greater that 8 bits. You can create a DirectDrawPalette object by calling the **IDirectDraw2::CreatePalette** method. DirectDrawPalette objects expose their functionality through the **IDirectDrawPalette** interface. For more information, see Palettes.

DirectDrawClipper object
The DirectDrawClipper object (casually referred to as a "clipper") helps you prevent blitting to certain portions of a surface or beyond the bounds of a surface. You can create a clipper by calling the **IDirectDraw2::CreateClipper** method. DirectDrawClipper objects expose their functionality through the **IDirectDrawClipper** interface. For more information, see Clippers.

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

Hardware Abstraction Layer (HAL)

DirectDraw provides device independence through the hardware-abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that DirectDraw uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, DirectDraw exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL in a combination of 16-bit and 32-bit code under Windows 95. Under Windows NT, the HAL is always implemented in 32-bit code. The HAL can be part of the display driver or a separate DLL that communicates with the display driver through a private interface that driver's creator defines.

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

Software Emulation

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

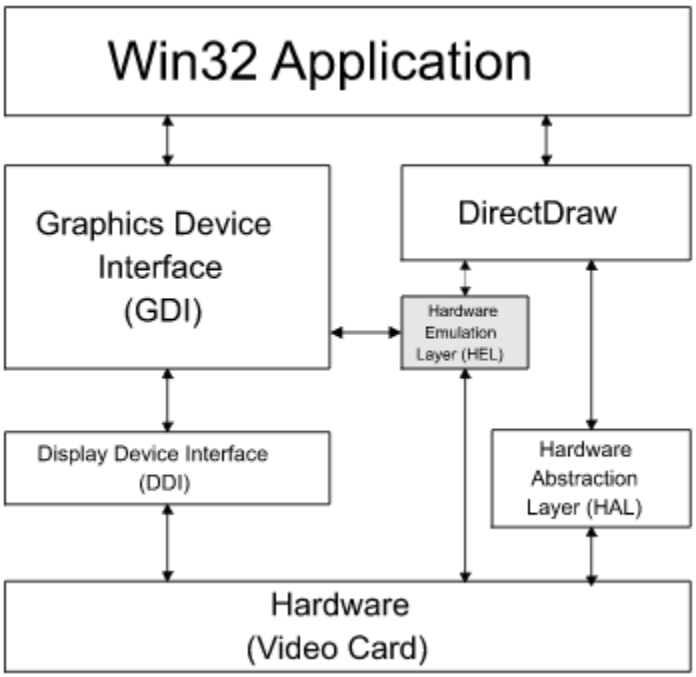
Obviously, software emulation cannot equal the performance that hardware features provide. You can query for the features the hardware supports by using the **IDirectDraw2::GetCaps** method. By examining these capabilities during application initialization, you can adjust application parameters to provide optimum performance over varying levels of hardware performance.

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

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

System Integration

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



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

DirectDraw Essentials

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

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

Cooperative Levels

Cooperative levels describe how DirectDraw interacts with the display and how it reacts to events that might affect the display. Use the **IDirectDraw2::SetCooperativeLevel** method to set cooperative level of DirectDraw. For the most part, you use DirectDraw cooperative levels to determine whether your application runs as a full screen program with exclusive access to the display or as a windowed application. However, DirectDraw cooperative levels can also have the following effects:

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

The normal cooperative level indicates that your DirectDraw application will operate as a windowed application. At this cooperative level you won't be able to change the primary surface's palette or perform page flipping. Additionally, you won't be able to call some methods that drastically affect the display or video memory, such as **IDirectDraw2::Compact**.

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

Because applications can use DirectDraw with multiple windows, **IDirectDraw2::SetCooperativeLevel** does not require a window handle to be specified if the application is requesting the DDSCL_NORMAL mode. By passing a NULL to the window handle, all of the windows can be used simultaneously in normal Windows mode.

IDirectDraw2::SetCooperativeLevel maintains a binding between a process and a window handle. If **IDirectDraw2::SetCooperativeLevel** is called once in a process, a binding is established between the process and the window. If it is called again in the same process with a different non-null window handle, it returns the DDERR_HWNDALREADYSET error value. Some applications may receive this error value when DirectSound® specifies a different window handle than DirectDraw—they should specify the same, top-level application window handle.

Display Modes

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

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

About Display Modes

A display mode is a hardware setting that describes the dimensions and bit-depth of graphics that the display hardware sends to the monitor from the primary surface. Display modes are described by their defining characteristics: width, height, and bit-depth. For instance, most display adapters can display graphics 640 pixels wide and 480 pixels tall, where each pixel is 8 bits of data. In shorthand, this display mode could be called 640-by-480-by-8 (640x480x8). As the dimensions of a display mode get larger or as the bit-depth increases, more display memory is required.

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

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

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

Determining Supported Display Modes

Because display hardware varies, not all devices will support all display modes. To determine the display modes supported on a given system, call the **IDirectDraw2::EnumDisplayModes** method. By setting the appropriate values and flags, the **IDirectDraw2::EnumDisplayModes** method can list all supported display modes or confirm that a single display mode that you specify is supported. The method's first parameter, *dwFlags*, controls extra options for the method; in most cases, you will set *dwFlags* to 0 to ignore extra options. The second parameter, *lpDDSurfaceDesc*, is the address of a **DDSURFACEDESC** structure that describes a given display mode to be confirmed; you'll usually set this parameter to NULL to request that all modes be listed. The third parameter, *lpContext*, is a pointer that you want DirectDraw to pass to your callback function; if you don't need any extra data in the callback function, use NULL here. Last, you set the *lpEnumModesCallback* parameter to the address of the callback function that DirectDraw will call for each supported mode.

The callback function you supply when calling **IDirectDraw2::EnumDisplayModes** must match the prototype for the **EnumModesCallback** function. For each display mode that the hardware supports, DirectDraw calls your callback function passing two parameters. The first parameter is the address of a **DDSURFACEDESC** structure that describes one supported display mode, and the second parameter is the address of the application-defined data you specified when calling **IDirectDraw2::EnumDisplayModes**, if any.

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

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

Setting Display Modes

You can set the display mode by using the **IDirectDraw2::SetDisplayMode** method. The **SetDisplayMode** method accepts four parameters that describe the dimensions, bit depth, and refresh rate of the mode to be set. The method uses a fifth parameter to indicate special options for the given mode; this is currently only used to differentiate between Mode 13 and the Mode X 320x200x8 display mode.

Although you can specify the desired display mode's bit depth, you cannot specify the pixel format that the display hardware will use for that bit depth. To determine the RGB bit masks that the display hardware uses for the current bit depth, call **IDirectDraw2::GetDisplayMode** after setting the display mode. If the current display mode is not palettized, you can examine the mask values in the **dwRBitMask**, **dwGBitMask**, and **dwBBitMask** members to determine the correct red, green, and blue bits. For more information, see Pixel Format Masks.

Modes can be changed by more than one application as long as they are all sharing a display card. You can change the bit depth of the display mode only if your application has exclusive access to the DirectDraw object. All DirectDrawSurface objects lose surface memory and become inoperative when the mode is changed. A surface's memory must be reallocated by using the **IDirectDrawSurface3::Restore** method.

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

Restoring Display Modes

You can explicitly restore the display hardware to its original mode by calling the **IDirectDraw2::RestoreDisplayMode** method. If the display mode was set by calling **IDirectDraw2::SetDisplayMode** (rather than **IDirectDraw::SetDisplayMode**) and your application takes the exclusive cooperative level, the original display mode is reset automatically when you set the application's cooperative level back to normal. If you're using the **IDirectDraw** interface, you must always explicitly restore the display mode.

Mode X and Mode 13 Display Modes

DirectDraw supports both Mode 13 and Mode X display modes. Mode 13 is the linear unflippable 320x200 8 bits per pixel palettized mode known widely by its hexadecimal BIOS mode number: 13. For more information, see Mode 13 Support. *Mode X* is a hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256 kilobytes (KB) of display memory (rather than the 64 KB allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system.

On Windows 95, DirectDraw provides two Mode X modes (320×200×8 and 320×240×8) for all display cards. Some cards also support linear low-resolution modes. In linear low-resolution modes, the primary surface can be locked and directly accessed. This is not possible in Mode X modes.

Mode X modes are available only if an application uses the DDSCL_ALLOWMODEX, DDSCL_FULLSCREEN, and DDSCL_EXCLUSIVE flags when calling the **IDirectDraw2::SetCooperativeLevel** method. If DDSCL_ALLOWMODEX is not specified, the **IDirectDraw2::EnumDisplayModes** method will not enumerate Mode X modes, and the **IDirectDraw2::SetDisplayMode** method will fail if a Mode X mode is requested.

Windows 95 and Windows NT do not support Mode X modes; therefore, when your application is in a Mode X mode, you cannot use the **IDirectDrawSurface3::Lock** or **IDirectDrawSurface3::Blt** methods to lock or blit to the primary surface. You also cannot use either the **IDirectDrawSurface3::GetDC** method on the primary surface, or GDI with a screen DC. Mode X modes are indicated by the DDSCAPS_MODEX flag in the **DDSCAPS** structure, which is part of the **DDSURFACEDESC** structure returned by the **IDirectDrawSurface3::GetCaps** and **IDirectDraw2::EnumDisplayModes** methods.

Mode X modes and some linear low-resolution modes are not supported on Windows NT.

Support for High Resolutions and True-Color Bit Depths

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

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

Window 95 and Windows NT allow you to specify the type of monitor being used. DirectDraw checks a list of known display modes against the display restrictions of the installed monitor. If DirectDraw determines that the requested mode is not compatible with the monitor, the call to the **IDirectDraw2::SetDisplayMode** method fails. Only modes that are supported on the installed monitor will be enumerated when you call the **IDirectDraw2::EnumDisplayModes** method.

The DirectDraw Object

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

- What Are DirectDraw Objects?
- What's New in IDirectDraw2?
- Cooperative Levels
- Display Modes
- Multiple DirectDraw Objects per Process
- Creating DirectDraw Objects by Using CoCreateInstance

What Are DirectDraw Objects?

The DirectDraw object is the heart of all DirectDraw applications and is an integral part of Direct3D® applications as well. It is the first object you create and, through it, you create all other related objects. Typically, you create a DirectDraw object by calling the **DirectDrawCreate** function, which returns an **IDirectDraw** interface. If you want to work with a different iteration of the interface (such as **IDirectDraw2**) to take advantage of new features it provides, you can query for it. Note that you can create multiple DirectDraw objects, one for each display device installed in a system.

The DirectDraw object represents the display device and makes use of hardware acceleration if the display device for which it was created supports hardware acceleration. Each unique DirectDraw object can manipulate the display device and create surfaces, palettes, and clipper objects that are dependent on (or are, "connected to") the object that created them. For example, to create surfaces, you call the **IDirectDraw2::CreateSurface** method. Or, if you need a palette object to apply to a surface, call the **IDirectDraw2::CreatePalette** method. Additionally, the **IDirectDraw2** interface exposes similar methods to create clipper objects.

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

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

What's New in IDirectDraw2?

This section details new features provided by the **IDirectDraw2** interface and describes how it behaves differently than its predecessor, **IDirectDraw**. The following topics are discussed:

- New Features in IDirectDraw2
- Cooperative Levels and Display Modes with IDirectDraw2
- Getting an IDirectDraw2 Interface

New Features in IDirectDraw2

The **IDirectDraw2** interface extends the IDirectDraw interface by adding the **IDirectDraw2::GetAvailableVidMem** method. This method enables you to query the display hardware for information about the status of its total available video memory and how much of that memory is available to be used by a surface of a given type.

Cooperative Levels and Display Modes with IDirectDraw2

The interaction between the **IDirectDraw2::SetCooperativeLevel** and **IDirectDraw2::SetDisplayMode** methods differs from that of their **IDirectDraw** counterparts.

If your application uses the **IDirectDraw** interface to set the full-screen exclusive cooperative level and change the display mode, the display mode will not be automatically restored when you return to the normal cooperative level-you have to call the **IDirectDraw::RestoreDisplayMode** method. However, if you use the **IDirectDraw2** interface, calling **RestoreDisplayMode** isn't necessary. However, the **IDirectDraw2::RestoreDisplayMode** method is supported for applications that want to explicitly restore the original display mode.

Getting an IDirectDraw2 Interface

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

The preceding example creates a DirectDraw object, then calls the **IUnknown::QueryInterface** method of the **IDirectDraw** interface it received to create an **IDirectDraw2** interface.

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

Multiple DirectDraw Objects per Process

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

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

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

Creating DirectDraw Objects by Using CoCreateInstance

You can create a DirectDraw object by using the **CoCreateInstance** function and the **IDirectDraw2::Initialize** method rather than the **DirectDrawCreate** function. The following steps describe how to create the DirectDraw object:

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

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

2. Create the DirectDraw object by using **CoCreateInstance** and the **IDirectDraw2::Initialize** method.

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

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

3. Before you use the DirectDraw object, you must call **IDirectDraw2::Initialize**. This method takes the driver GUID parameter that the **DirectDrawCreate** function typically uses (NULL in this case). After the DirectDraw object is initialized, you can use and release it as if it had been created by using the **DirectDrawCreate** function. If you do not call the **IDirectDraw2::Initialize** method before using one of the methods associated with the DirectDraw object, a DDERR_NOTINITIALIZED error will occur.

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

```
CoUnitialize();
```

Surfaces

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

- Basic Concepts
- Creating Surfaces
- Flipping Surfaces
- Losing Surfaces
- Releasing Surfaces
- Updating Surface Characteristics
- Accessing the Frame-Buffer Directly
- Using Non-local Video Memory Surfaces
- Converting Color and Format
- Overlay Surfaces
- Blitting to Multiple Windows

Basic Concepts

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

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

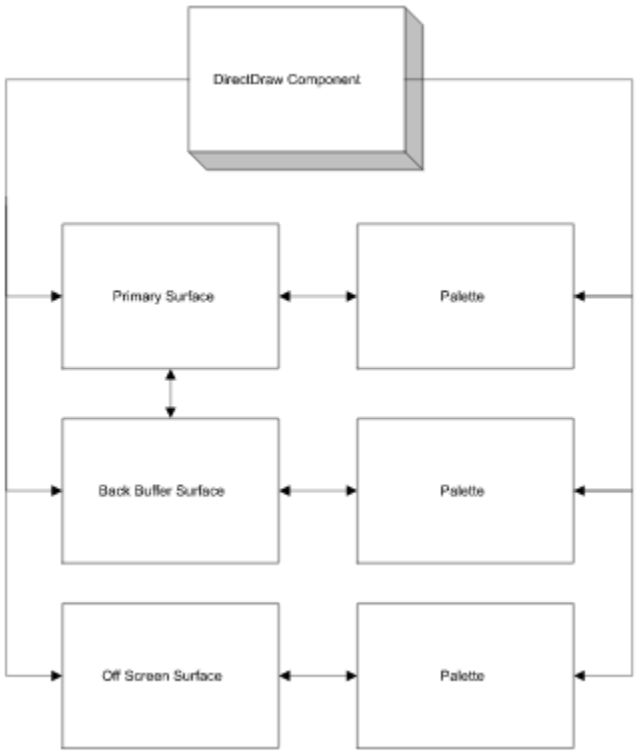
What Are Surfaces?

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

Using the **IDirectDraw2::CreateSurface** method, you can create a single surface object, complex surface-flipping chains, or three-dimensional surfaces. The **CreateSurface** method creates the requested surface or flipping chain and retrieves a pointer to the primary surface's **IDirectDrawSurface** interface through which the object exposes its functionality. If you want to work with a different iteration of the interface (such as **IDirectDrawSurface3**), you can query for it.

The **IDirectDrawSurface3** interface enables you to indirectly access memory through blit methods, such as **IDirectDrawSurface3::BltFast**. The surface object can provide a device context to the display that you can use with GDI functions. Additionally, you can use **IDirectDrawSurface3** methods to directly access display memory. For example, you can use the **IDirectDrawSurface3::Lock** method to lock the display memory and retrieve the address corresponding to that surface. Addresses of display memory might point to visible frame buffer memory (primary surface) or to nonvisible buffers (off-screen or overlay surfaces). Nonvisible buffers usually reside in display memory, but can be created in system memory if required by hardware limitations or if DirectDraw is performing software emulation. In addition, the **IDirectDrawSurface3** interface extends other methods that you can use to set or retrieve palettes, or to work with specific types or surfaces, like flipping chains or overlays.

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



Surface Interfaces

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

The **IDirectDrawSurface** interface is the oldest version of the interface and is provided by default when you create a surface by using the **IDirectDraw2::CreateSurface** method. To utilize the new functionality provided by another version of the interface, you must query for the new version by calling its **QueryInterface** method. The following example shows how you can do this:

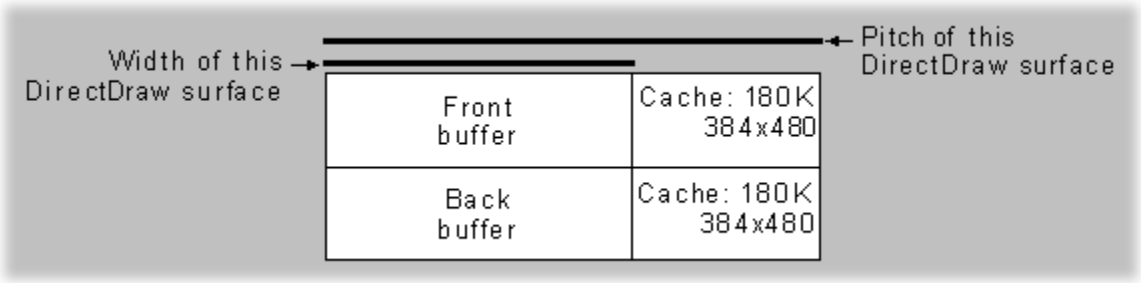
```
LPDIRECTDRAWSURFACE  lpSurf;
LPDIRECTDRAWSURFACE2 lpSurf2;

// 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 preceding example retrieves a DirectDrawSurface object's **IDirectDrawSurface2** interface by specifying the IID_IDirectDraw2 reference identifier when it calls the **QueryInterface** method. To retrieve an **IDirectDrawSurface3** interface, use the IID_IDirectDrawSurface3 reference identifier instead.

Width and Pitch

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

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



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

Therefore, when rendering directly into surface memory, always use the pitch returned by the **IDirectDrawSurface3::Lock** method (or the **IDirectDrawSurface3::GetDC** method). Do not assume a pitch based solely on the display mode. If your application works on some display adapters but looks garbled on others, this may be the cause of your problem.

Color Keying

DirectDraw supports source and destination *color keying* for blits and overlay surfaces. You can supply a color key or a color range for both of these types of color keying. For general information about color keying, see Transparent Blitting and Color Keys. You set a surface's color key by calling the its **IDirectDrawSurface3::SetColorKey** method.

When blitting, *source color keying* specifies a color or color range that is not copied. Likewise, *destination color keying* specifies a color or color range that is replaced The source color key specifies what can and cannot be read from the surface. The destination color key specifies what can and cannot be written onto, or covered up, on the destination surface. If a destination surface has a color key, only the pixels that match the color key are changed, or covered up, on the destination surface.

In addition to blit-related color keys, overlay surfaces can use overlay color keys. For more information, see Overlay Color Keys.

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

Color keys are specified using the pixel format of a surface. If a surface is in a palettized format, the color key is specified as an index or a range of indices. If the surface's pixel format is specified by a FOURCC code that describes a YUV format, the YUV color key is specified by the three low-order bytes in both the **dwColorSpaceLowValue** and **dwColorSpaceHighValue** members of the **DDCOLORKEY** structure. The lowest order byte contains the V data, the second lowest order byte contains the U data, and the highest order byte contains the Y data. The *dwFlags* parameter of the **IDirectDrawSurface3::SetColorKey** method specifies whether the color key is to be used for overlay or blit operations, and whether it is a source or a destination key. Some examples of valid color keys follow:

8-bit palettized mode

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

24-bit true-color mode

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

FourCC YUV mode

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

Pixel Formats

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

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

You can retrieve information about an existing surface's pixel format by calling the **IDirectDrawSurface3::GetPixelFormat** method.

Creating Surfaces

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

Use the **IDirectDraw2::CreateSurface** method to create one surface or to simultaneously create multiple surfaces (a complex surface). When calling **CreateSurface**, you specify the dimensions of the surface, whether it is a single surface or a complex surface, and the pixel format (if the surface won't be using an indexed palette). All these characteristics are contained in a **DDSURFACEDESC** structure, whose address you send with the call. If the hardware can't support the requested capabilities or if it previously allocated those resources to another DirectDrawSurface object, the call will fail.

Creating single surfaces or multiple surfaces is a simple matter that requires only a few lines of code. There are four main scenarios for creating surfaces. Each scenario requires a little more preparation than the one before it, but none are difficult. The following four scenarios are discussed:

1. Creating the Primary Surface
2. Creating an Off-Screen Surface
3. Creating Complex Surfaces and Flipping Chains
4. Creating Wide Surfaces

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

Creating the Primary Surface

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

When you create a primary surface, the dimensions implicitly match the current display mode. Therefore, this is the one time you don't need to declare surface dimensions. Frankly, if you do specify them, the call fails-even if they match the current display mode.

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

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

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

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

Creating an Off-Screen Surface

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

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

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

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

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

// Request a simple off-screen surface, sized
// 100 by 100 pixels.
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
dwHeight = 100;
dwWidth = 100;
```

In previous versions of DirectX, the maximum width of off-screen surfaces was limited to the width of the primary surface. With DirectX 5, you can create surfaces as wide as you need, permitting that the display hardware can support them. Be careful when declaring wide off-screen surfaces; if the video card memory cannot hold a surface as wide as you request, the surface is created in system memory. If you explicitly choose video memory and the hardware can't support it, the call fails.

Creating Complex Surfaces and Flipping Chains

You can also create complex surfaces. A complex surface is a set of surfaces created with a single call to the **IDirectDraw2::CreateSurface** method. If the DDSCAPS_COMPLEX flag is set when you call **CreateSurface** call, DirectDraw implicitly creates one or more surfaces in addition to the surface explicitly specified. You manage complex surfaces just like a single surface—a single call to the **IDirectDraw::Release** method releases all surfaces, and a single call to the **IDirectDrawSurface3::Restore** method restores them all. However, implicitly created surfaces cannot be detached. For more information, see **IDirectDrawSurface3::DeleteAttachedSurface**.

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

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

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

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

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

The previous example constructs a double-buffered flipping environment—a single call to the **IDirectDrawSurface3::Flip** method exchanges the surface memory of the primary surface and the back buffer. If you specify 2 for the value of the **dwBackBufferCount** member of the **DDSURFACEDESC** structure, two back buffers are created, and each call to **Flip** rotates the surfaces in a circular pattern, providing a triple-buffered flipping environment.

Creating Wide Surfaces

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

To check for wide surface support, call **IDirectDraw2::GetCaps** and look for the DDCAPS2_WIDESURFACES flag in the **dwCaps2** member of the first **DDCAPS** structure you send with the call. If the flag is present, you can create video memory off-screen surfaces that are wider that the primary surface.

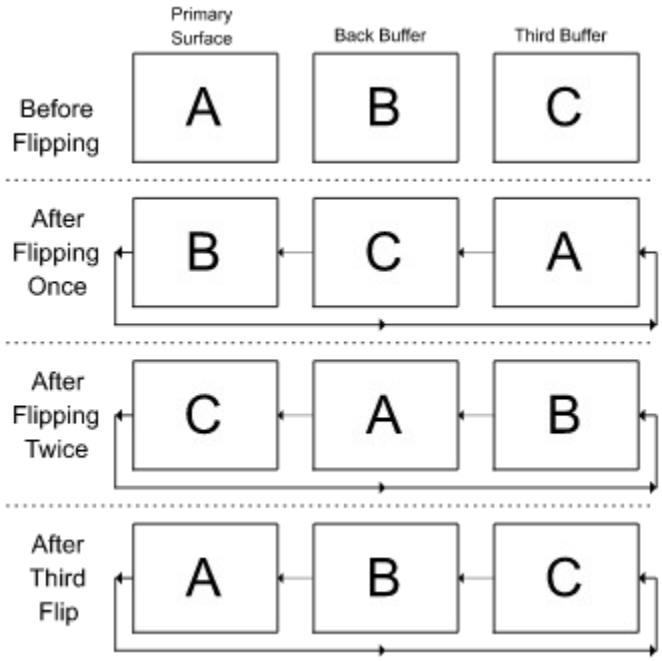
If you attempt to create a wide surface in video memory when the DDCAPS2_WIDESURFACES flag isn't present, the attempt will fail and return DDERR_INVALIDPARAMS.

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

Flipping Surfaces

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

Typically, when you use the **IDirectDrawSurface3::Flip** method to request a surface flip operation, the pointers to surface memory for the primary surface and back buffers are swapped. Flipping is performed by switching pointers that the display device uses for referencing memory, not by copying surface memory. (The exception to this is when DirectDraw is emulating the flip, in which case it simply copies the surfaces. DirectDraw emulates flip operations if a back buffer cannot fit into display memory or if the hardware doesn't support DirectDraw.) When a flipping chain contains a primary surface and more than one back-buffer, the pointers are switched in a circular pattern, as shown in the following illustration:



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

Remember, DirectDraw flips surfaces by swapping surface memory pointers within DirectDrawSurface objects, not by swapping the objects themselves. This means that, to blit to the back buffer in any type of flipping scheme, you always use the same DirectDrawSurface object-the one that was the back buffer when you created the flipping chain. Conversely, you always perform a flip operation by calling the front surface's **Flip** method.

When working with visible surfaces, such as a primary surface flipping chain or a visible overlay surface flipping chain, the **Flip** method is asynchronous unless you include the DDFLIP_WAIT flag. On these visible surfaces, the **Flip** method can return before the actual flip operation occurs in the hardware (because the hardware doesn't flip until the next vertical refresh occurs). While the actual flip operation is pending, the back buffer behind the currently visible surface can't be locked or blitted by calling the **IDirectDrawSurface3::Lock**, **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltFast**, or **IDirectDrawSurface3::GetDC** methods. If you attempt to call these methods while a flip operation is pending, they will fail and return DDERR_WASSTILLDRAWING. However, if you are using a triple buffered scheme, the rearmost buffer is still available.

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 return DDERR_SURFACELOST and perform no other action.

Surfaces can be lost because the display card mode was changed or because another application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. The **IDirectDrawSurface3::Restore** method re-creates these lost surfaces and reconnects them to their DirectDrawSurface object. Restoring a surface doesn't reload any bitmaps that may have existed in the surface prior to its loss. Therefore, if you lose a surface you must also completely reconstitute the graphics it once held.

For more information, see Setting Display Modes.

Releasing Surfaces

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

Each surface you individually create must be explicitly released. However, if you implicitly created multiple surfaces with a single call to **IDirectDraw2::CreateSurface** or **IDirectDraw::CreateSurface**, such as a flipping chain, you need only release the front buffer. In this case, any pointers you might have to back buffer surfaces are implicitly released and can no longer be used.

Updating Surface Characteristics

You can update the characteristics of an existing surface by using the **IDirectDrawSurface3::SetSurfaceDesc** method. With this method, you can change the pixel format and location of a DirectDrawSurface object's surface memory to system memory that your application has explicitly allocated. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client application must also deallocate it. For more information about how this method is used, see Updating Surface Characteristics.

When calling the **IDirectDrawSurface3::SetSurfaceDesc** method, the *lpddsd* parameter must be the address of a **DDSURFACEDESC** structure that describes the new surface memory as well as a pointer to that memory. Within the structure, you can only set the **dwFlags** member to reflect valid members for the location of the surface memory, dimensions, pitch, and pixel format. Therefore, **dwFlags** can only contain combinations of the DDSD_WIDTH, DDSD_HEIGHT, DDSD_PITCH, DDSD_LPSURFACE, and DDSD_PIXELFORMAT flags, which you set to indicate valid structure members.

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

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

There are some restrictions you must be aware of when using **IDirectDrawSurface3::SetSurfaceDesc**, some of which are common sense. For example, the **lpSurface** member of the **DDSURFACEDESC** structure must be a valid pointer to a system memory (the method doesn't support video memory pointers at this time). Also, the **dwWidth** and **dwHeight** members must be nonzero values. Lastly, you cannot reassign the primary surface or any surfaces within the primary's flipping chain.

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

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

Accessing the Frame-Buffer Directly

You can directly access surface memory in the frame-buffer or in system memory by using the **IDirectDrawSurface3::Lock** method. When you call this method, the *lpDestRect* parameter is a pointer to a **RECT** structure that describes the rectangle on the surface you want to access directly. To request that the entire surface be locked, set *lpDestRect* to NULL. Also, you can specify a **RECT** that covers only a portion of the surface. Providing that no two rectangles overlap, two threads or processes can simultaneously lock multiple rectangles in a surface.

The **Lock** method fills a **DDSURFACEDESC** structure with all the information you need to properly access the surface memory. The structure includes information about the pitch (or stride) and the pixel format of the surface, if different from the pixel format of the primary surface. When you finish accessing the surface memory, call the **IDirectDrawSurface3::Unlock** method to unlock it.

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

- Never assume a constant display pitch. Always examine the pitch information returned by the **IDirectDrawSurface3::Lock** method. This pitch can vary for a number of reasons, including the location of the surface memory, the type of display card, or even the version of the DirectDraw driver. For more information, see Width and Pitch.
- Make certain you blit to unlocked surfaces. DirectDraw blit methods will fail, returning DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES, if called on a locked surface. Similarly, GDI blit functions fail without returning error values if called on a locked surface that exists in display memory.
- Limit your application's activity while a surface is locked. While a surface is locked, DirectDraw often holds the Win16Lock so that gaining access to surface memory can occur safely. The Win16Lock serializes access to GDI and USER, shutting down Windows for the duration between the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** calls. The **IDirectDrawSurface3::GetDC** method implicitly calls **IDirectDrawSurface3::Lock**, and the **IDirectDrawSurface3::ReleaseDC** implicitly calls **IDirectDrawSurface3::Unlock**.
- Copy aligned to display memory. Windows 95 uses a page fault handler, Vflatd.386, to implement a virtual flat-frame buffer for display cards with bank-switched memory. The handler allows these display devices to present a linear frame buffer to DirectDraw. Copying unaligned to display memory can cause the system to suspend operations if the copy spans memory banks.

Locking the surface typically causes DirectDraw to take the Win16Lock. During the Win16Lock all other applications, including Windows, cease execution. Since the Win16Lock stops applications from executing, standard debuggers cannot be used while the lock is held. Kernel debuggers can be used during this period.

If a blit is in progress when you call **IDirectDrawSurface3::Lock**, the method will return immediately with an error, as a lock cannot be obtained. To prevent the error, use the DDLOCK_WAIT flag to cause the method to wait until a lock can be successfully obtained.

Using Non-local Video Memory Surfaces

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

Currently, there are two implementations of the AGP architecture, known as the “execute model” and the “DMA model.” In the execute model implementation, the display device supports the same features for non-local video memory surfaces and local video memory surfaces. As a result, when you retrieve hardware capabilities by calling the **IDirectDraw2::GetCaps** method, the blit-related flags in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure will be identical to those for local video memory. Under the execute model, if local video memory is exhausted, DirectDraw will automatically fall back on non-local video memory unless the caller specifically requests otherwise.

In the DMA model implementation, support for blitting and texturing from non-local video memory surfaces is limited. When the display device uses the DMA model, the DDCAPS2_NONLOCALVIDMEMCAPS flag will be set in the **dwCaps2** member when you retrieve device capabilities. In the DMA model, the blit-related flags included in the **dwNLVBCaps**, **dwNLVBCaps2**, **dwNLVBCKeyCaps**, **dwNLVBFXCaps**, and **dwNLVBRops** members of the **DDCAPS** structure describe the features that are supported; these features will often be a smaller subset of those supported for local video memory surfaces. Under the DMA model, DirectDraw will never create a surface in non-local video memory unless the caller explicitly requests it.

DMA model implementations vary in support for texturing from non-local video memory surfaces. If the driver supports texturing from non-local video memory surfaces, the D3DDEVCAPS_TEXTURENONLOCALVIDMEM flag will be set when you retrieve the 3-D device’s capabilities by calling the **IDirect3DDevice2::GetCaps** method.

Converting Color and Format

Non-RGB surface formats are described by four-character codes (FOURCC codes). If an application calls the **IDirectDrawSurface3::GetPixelFormat** method to request the pixel format, and the surface is a non-RGB surface, the DDPF_FOURCC flag will be set and the **dwFourCC** member of the **DDPIXELFORMAT** structure will be valid. If the FOURCC code represents a YUV format, the DDPF_YUV flag will also be set and the **dwYUVBitCount**, **dwYBits**, **dwUBits**, **dwVBits**, and **dwYUVAlphaBits** members will be valid masks that can be used to extract information from the pixels.

If an RGB format is present, the DDPF_RGB flag will be set and the **dwRGBBitCount**, **dwRBits**, **dwGBits**, **dwBBits**, and **dwRGBAlphaBits** members will be valid masks that can be used to extract information from the pixels. The DDPF_RGB flag can be set in conjunction with the DDPF_FOURCC flag if a nonstandard RGB format is being described.

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

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

Overlay Surfaces

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

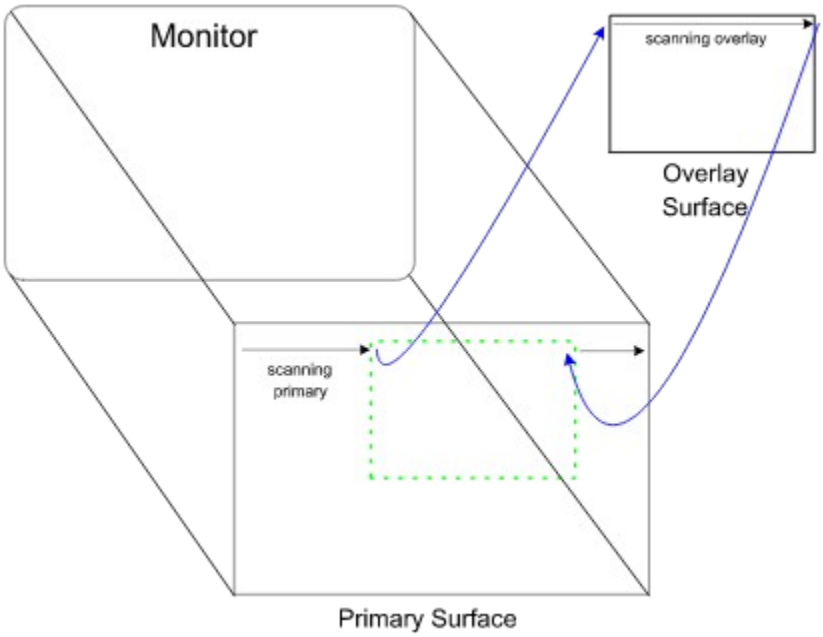
- Overlay Surface Overview
- Significant **DDCAPS** Members and Flags
- Source and Destination Rectangles
- Boundary and Size Alignment
- Minimum and Maximum Stretch Factors
- Overlay Color Keys
- Positioning Overlay Surfaces
- Creating Overlay Surfaces
- Overlay Z-Orders
- Flipping Overlay Surfaces

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

Overlay Surface Overview

Overlay surfaces, casually referred to as overlays, are surfaces with special hardware supported capabilities. Overlay surfaces are frequently used to display live video, recorded video, or still bitmaps over the primary surface without blitting to the primary surface or changing the primary surface's contents in any way. Overlay surface support is provided entirely by the hardware; DirectDraw supports any capabilities as reported by the display device driver. DirectDraw does not emulate overlay surfaces.

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



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

You create overlay surfaces by calling the **IDirectDraw2::CreateSurface** method, specifying the DDSCAPS_OVERLAY flag in the associated **DDSCAPS** structure. Overlay surfaces can only be created in video memory, so you must also include the DDSCAPS_VIDEOMEMORY flag. As with other types of surfaces, by including the appropriate flags you can create either a single overlay or a flipping chain made up of multiple overlay surfaces.

Significant DDCAPS Members and Flags

You can retrieve information about the supported overlay features by calling the **IDirectDraw2::GetCaps** method. The method fills a **DDCAPS** structure with information describing all features.

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

Member	Flag
dwMaxVisibleOverlays	This member is always valid
dwCurrVisibleOverlays	This member is always valid
dwAlignBoundarySrc	DDCAPS_ALIGNBOUNDARYSRC
dwAlignSizeSrc	DDCAPS_ALIGNSIZESRC
dwAlignBoundaryDest	DDCAPS_ALIGNBOUNDARYDEST
dwAlignSizeDest	DDCAPS_ALIGNSIZEDEST
dwMinOverlayStretch	DDCAPS_OVERLAYSTRETCH
dwMaxOverlayStretch	DDCAPS_OVERLAYSTRETCH

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

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

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

Source and Destination Rectangles

To display an overlay surface, you call the overlay surface's **IDirectDrawSurface3::UpdateOverlay** method, specifying the DDOVER_SHOW flag in the *dwFlags* parameter. The method requires you to specify a source and destination rectangle in the *lpSrcRect* and *lpDestRect* parameters. The source rectangle describes a rectangle on the overlay surface that will be visible on the primary surface. To request that the method use the entire surface, set the *lpSrcRect* parameter to NULL. The destination rectangle describes a portion of the primary surface on which the overlay surface will be displayed.

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

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

Boundary and Size Alignment

Due to various hardware limitations, some device drivers impose restrictions on the position and size of the source and destination rectangles used to display overlay surfaces. To find out which restrictions apply for a device, call the **IDirectDraw2::GetCaps** method and then examine the overlay-related flags in the **dwCaps** member of the **DDCAPS** structure. The following table shows the members and flags specific to boundary and size alignment restrictions:

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

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

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

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

Minimum and Maximum Stretch Factors

Due to hardware limitations, some devices restrict how wide a destination rectangle can be compared with the corresponding source rectangle. DirectDraw communicates these restrictions as stretch factors. A stretch factor is the ratio between the widths of the source and destination rectangles. If the driver provides information about stretch factors, it sets the DDCAPS_OVERLAYSTRETCH flag in the **DDCAPS** structure after you call the **IDirectDraw2::GetCaps** method. Note that stretch factors are reported multiplied by 1000, so a value of 1300 actually means 1.3 (and 750 would be 0.75).

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

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

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

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

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

Overlay Color Keys

Like other types of surfaces, overlay surfaces use source and destination color keys for controlling transparent blit operations between surfaces. Because overlay surfaces are not displayed by blitting, there needs to be a different way to control how an overlay surface is displayed over the primary surface when you call the **IDirectDrawSurface3::UpdateOverlay** method. This need is filled by overlay color keys. Overlay color keys, like their blit-related counterparts, have a source version and a destination version that you set by calling the **IDirectDrawSurface3::SetColorKey** method. You use the DDCKEY_SRCOVERLAY or DDCKEY_DESTOVERLAY flags to set a source or destination overlay color key. Overlay surfaces can employ blit and overlay color keys together to control blit operations and overlay display operations appropriately; the two types of color keys do not conflict with one another.

The **IDirectDrawSurface3::UpdateOverlay** method uses the source overlay color key to determine which pixels in the overlay surface should be considered transparent, allowing the contents of the primary surface to show through. Likewise, the method uses the destination overlay color key to determine the parts of the primary surface that will be covered up by the overlay surface when it is displayed. The resulting visual effect is the same as that created by blit-related color keys. For more information, see Transparent Blitting and Color Keys and Color Keying.

Positioning Overlay Surfaces

After initially displaying an overlay by calling the **IDirectDrawSurface3::UpdateOverlay** method, you can update the destination rectangle's by calling the **IDirectDrawSurface3::SetOverlayPosition** method.

Make sure that the positions you specify comply with any boundary alignment restrictions enforced by the hardware. For more information, see Boundary and Size Alignment. Also remember that **IDirectDraw2::SetOverlayPosition** doesn't perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Creating Overlay Surfaces

Like all surfaces, you create an overlay surface by calling the **IDirectDraw2::CreateSurface** method. To create an overlay, include the DDSCAPS_OVERLAY flag in the associated **DDSCAPS** structure.

Overlay support varies widely across display devices. As a result, you cannot be sure that a given pixel format will be supported by most drivers and must therefore be prepared to work with a variety of pixel formats. You can request information about the non-RGB formats that a driver supports by calling the **IDirectDraw2::GetFourCCCodes** method.

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

You can create overlay surface flipping chains. For more information, see Creating Complex Surfaces and Flipping Chains.

Overlay Z-Orders

Overlay surfaces are assumed to be on top of all other screen components, but when you display multiple overlay surfaces, you need some way to visually organize them. DirectDraw supports *overlay z-ordering* to manage the order in which overlays clip each other. Z-order values represent conceptual distances from the primary surface toward the viewer. They range from 0, which is just on top of the primary surface, to 4 billion, which is as close to the viewer as possible, and no two overlays can share the same z-order. You set z-order values by calling the **IDirectDrawSurface3::UpdateOverlayZOrder** method.

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

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

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

Flipping Overlay Surfaces

Like other types of surfaces, you can create overlay flipping chains. After creating a flipping chain of overlays, call the **IDirectDrawSurface3::Flip** method to flip between them. For more information, see Flipping Surfaces.

Software decoders displaying video with overlay surfaces can use the DDFLIP_ODD and DDFLIP_EVEN flags when calling the **Flip** method to use features that reduce motion artifacts. If the driver supports odd-even flipping, the DDCAPS2_CANFLIPODDEVEN flag will be set in the **DDCAPS** structure after retrieving driver capabilities. If DDCAPS2_CANFLIPODDEVEN is set, you can include the DDOVER_BOB flag when calling the **IDirectDrawSurface3::UpdateOverlay** method to inform the driver that you want it to use the “Bob” algorithm to minimize motion artifacts. Later, when you call **Flip** with the DDFLIP_ODD or DDFLIP_EVEN flag, the driver will automatically adjust the overlay source rectangle to compensate for jittering artifacts.

If the driver doesn’t set the DDCAPS2_CANFLIPODDEVEN flag when you retrieve hardware capabilities, **UpdateOverlay** will fail if you specify the DDOVER_BOB flag.

For more information about the Bob algorithm, see Solutions to Common Video Artifacts.

Blitting to Multiple Windows

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

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

Palettes

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

What are Palettes?

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

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

You can create a palette by calling the **IDirectDraw2::CreatePalette** method. This method retrieves a pointer to the palette object's **IDirectDrawPalette** interface. You can use the methods of this interface to manipulate palette entries, retrieve information about the object's capabilities, or initialize the object (if you used the **CoCreateInstance** COM function to create it).

You apply a palette to a surface by calling the surface's **IDirectDrawSurface3::SetPalette** method. A single palette can be applied to multiple surfaces.

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

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

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

Palette Types

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

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

To create an indexed palette, specify the DDPCAPS_8BITENTRIES flag when calling the **IDirectDraw2::CreatePalette** method. For example, to create a 4-bit indexed palette, specify both the DDPCAPS_4BIT and DDPCAPS_8BITENTRIES flags. When you create an indexed palette, you pass a pointer to an array of bytes rather than a pointer to an array of **PALETTEENTRY** structures. You must cast the pointer to the array of bytes to an **LPPALETTEENTRY** type when you use the **IDirectDraw2::CreatePalette** method.

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

Setting Palettes on Nonprimary Surfaces

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

Nonprimary surface palettes are intended for use by Direct3D applications.

Sharing Palettes

Palettes can be shared among multiple surfaces. The same palette can be set on the front buffer and the back buffer of a flipping chain or shared among multiple texture surfaces. When an application attaches a palette to a surface by using the **IDirectDrawSurface3::SetPalette** method, the surface increments the reference count of that palette. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached palette. In addition, if a palette is detached from a surface by using **IDirectDrawSurface3::SetPalette** with a NULL palette interface pointer, the reference count of the surface's palette will be decremented.

Note If **IDirectDrawSurface3::SetPalette** is called several times consecutively on the same surface with the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

Palette Animation

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

There are two methods for providing straightforward palette animation:

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

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

The second method requires two or more DirectDrawPalette objects. When using this method, you perform the animation by attaching one palette object after another to the surface object by calling the **IDirectDrawSurface3::SetPalette** method.

Neither method is hardware intensive, so feel free to use whichever technique you see fit for your application.

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

Clippers

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

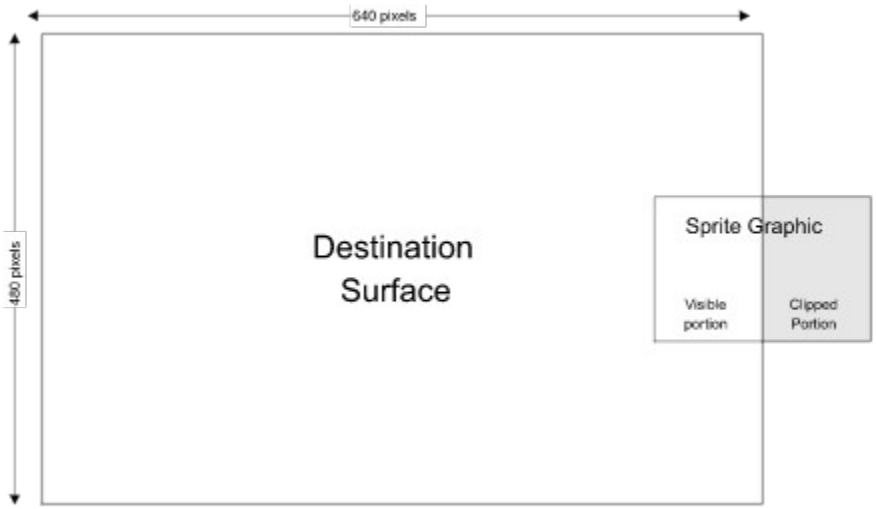
- What Are Clipper Objects?
- Clip Lists
- Sharing DirectDrawClipper Objects
- Independent DirectDrawClipper Objects
- Creating DirectDrawClipper Objects with CoCreateInstance
- Using a Clipper with the System Cursor
- Using a Clipper with Multiple Windows

What Are Clipper Objects?

Clippers, or DirectDrawClipper objects, allow you to blit to selected parts of a surface. A clipper object holds one or more clip lists. A clip list is one bounding rectangle or a list of several bounding rectangles that describe an area or areas of a surface to which you are allowed to blit. These areas are described with **RECT** structures, in screen coordinates.

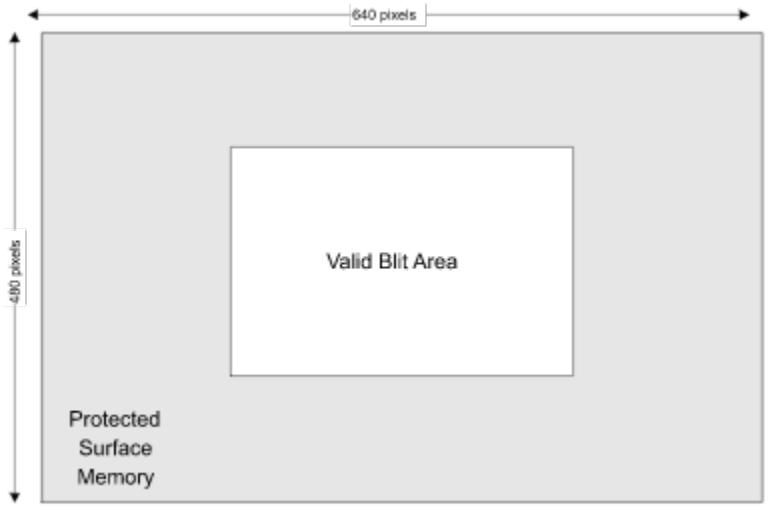
Clip lists are a very valuable tool. One common use for them is in preventing your application from blitting beyond the edges of the screen. For example, imagine that you want to display a sprite as it enters the screen from an edge. You don't want to make the sprite "pop" onto the screen; you want it to appear as though it is smoothly moving into view. Without a clipper object, you would need to include logic that restricts blit operations to protect surface memory that is logically off the edge of the screen. Failing to do this results in memory access violations.

The following illustration shows this type of clipping.



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

The following illustration shows this clipping style.



Clip Lists

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

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

Through the **IDirectDrawClipper::SetClipList** method, you can passing an entire clip list to the driver (if the driver supports this) rather than calling the driver multiple times, once for each rectangle in the clip list. Additionally, you can set the clipper to a single window by calling the **IDirectDrawClipper::SetHWnd** method, specifying the target window's handle. If you set a clipper using a window handle, you cannot set additional rectangles.

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

Sharing DirectDrawClipper Objects

DirectDrawClipper objects can be shared between multiple surfaces. For example, the same DirectDrawClipper object can be set on both the front buffer and the back buffer of a flipping chain. When an application attaches a DirectDrawClipper object to a surface by using the **IDirectDrawSurface3::SetClipper** method, the surface increments the reference count of that object. When the reference count of the surface reaches 0, the surface will decrement the reference count of the attached DirectDrawClipper object. In addition, if a DirectDrawClipper object is detached from a surface by calling **IDirectDrawSurface3::SetClipper** with a NULL clipper interface pointer, the reference count of the surface's DirectDrawClipper object will be decremented.

Note If **IDirectDrawSurface3::SetClipper** is called several times consecutively on the same surface for the same DirectDrawClipper object, the reference count for the object is incremented only once. Subsequent calls do not affect the object's reference count.

Independent DirectDrawClipper Objects

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

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

You can still create DirectDrawClipper objects by using the **IDirectDraw2::CreateClipper** method. These DirectDrawClipper objects are automatically released when the DirectDraw object from which they were created is released.

Creating DirectDrawClipper Objects with CoCreateInstance

DirectDrawClipper objects have full class-factory support for COM compliance. In addition to using the standard **DirectDrawCreateClipper** function and **IDirectDraw2::CreateClipper** method, you can also create a DirectDrawClipper object either by using the **CoGetClassObject** function to obtain a class factory and then calling the **CoCreateInstance** function, or by calling **CoCreateInstance** directly. The following example shows how to create a DirectDrawClipper object by using **CoCreateInstance** and the **IDirectDrawClipper::Initialize** method.

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

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

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

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

Using a Clipper with the System Cursor

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

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

A DirectDrawClipper object can prevent these artifacts from appearing by preventing the mouse cursor image from "being in the way" during a blit operation. It's a relatively simple matter to implement, as well. To do so, create a DirectDrawClipper object by calling the **IDirectDraw2::CreateClipper** method. Then, assign your application's window handle to the clipper with the **IDirectDrawClipper::SetHWnd** method. Once a clipper is attached, any subsequent blits you perform on the primary surface with the **IDirectDrawSurface3::Blt** method will not exhibit the artifact.

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

Using a Clipper with Multiple Windows

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

To do this, create a single DirectDraw object with a primary surface. Then, create a DirectDrawClipper object and assign it to your primary surface by calling the **IDirectDrawSurface3::SetClipper** method. To blit only the client area of a window, set the clipper to that window's client area by calling the **IDirectDrawClipper::SetHWnd** method before blitting to the primary surface. Whenever you need to blit to another window's client area, call the **IDirectDrawClipper::SetHWnd** method again with the new target window handle.

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

Advanced DirectDraw Topics

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

- Mode 13 Support
- Taking Advantage of DMA Support
- Using DirectDraw Palettes in Windowed Mode
- Working with Multiple Monitors
- Video-Ports
- Getting the Flip and Blit Status
- Blitting with Color Fill
- Determining the Capabilities of the Display Hardware
- Storing Bitmaps in Display Memory
- Triple Buffering
- DirectDraw Applications and Window Styles
- Matching True RGB Colors to the Frame Buffer’s Color Space

Mode 13 Support

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

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

About Mode 13

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

Setting Mode 13

Mode 13 has similar enumeration and mode-setting behavior as Mode X. DirectDraw will only enumerate Mode 13 if the DDSCL_ALLOWMODEX flag was passed to the **IDirectDraw2::SetCooperativeLevel** method.

You enumerate the Mode 13 display mode like all other modes, but you make a surface capabilities check before calling **IDirectDraw2::EnumDisplayModes**. To do this, call **IDirectDraw2::GetCaps** and check for the DDSCAPS_STANDARDVGAMODE flag in the **DDSCAPS** structure after the method returns. If this flag is not present, then Mode 13 is not supported, and attempts to enumerate with the DDEDM_STANDARDVGAMODES flag will fail, returning **DDERR_INVALIDPARAMS**.

The **EnumDisplayModes** method now supports a new enumeration flag, DDEDM_STANDARDVGAMODES, which causes DirectDraw to enumerate Mode 13 in addition to the 320x200x8 Mode X mode. There is also a new **IDirectDraw2::SetDisplayMode** flag, DDSDM_STANDARDVGAMODE, which you must pass in order to distinguish Mode 13 from 320x200x8 Mode X.

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

Mode 13 and Surface Capabilities

When DirectDraw calls an application's **EnumModesCallback** callback function, the **ddsCaps** member of the associated **DDSURFACEDESC** structure contains flags that reflect the mode being enumerated. You can expect DDSCAPS_MODEX for a Mode X mode or DDSCAPS_STANDARDVGAMODE for Mode 13. These flags are mutually exclusive. If neither of these bits is set, then the mode is a linear accelerated mode. This behavior also applies to the flags retrieved by the **IDirectDraw2::GetDisplayMode** method.

Using Mode 13

Because Mode 13 is a linear mode, DirectDraw can give an application direct access to the frame buffer. Unlike Mode X modes, you can call the **IDirectDrawSurface3::Lock**, **IDirectDrawSurface3::Blt**, and **IDirectDrawSurface3::BltFast** methods directly to access the primary surface.

When using Mode 13, DirectDraw supports an emulated **IDirectDrawSurface3::Flip** that is implemented as a straight copy of the contents of a back buffer to the primary surface. You can emulate this yourself by copying a smaller subrectangle of the back buffer to the primary using **Blt** or **BltFast**.

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

Taking Advantage of DMA Support

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

About DMA Device Support

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

Testing For DMA Support

Before using DMA operations, you must test the device for DMA support and, if it does support DMA, how much support it provides. Begin by retrieving the driver capabilities by calling the **IDirectDraw2::GetCaps** method, then look for the DDCAPS_CANBLTSYSMEM flag in the **dwCaps** member of the associated **DDCAPS** structure. If the flag is set, the device supports DMA.

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

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

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

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

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

Typical Scenarios For DMA

System memory to video memory SRCCOPY transfers are the most common type of hardware-supported blit operation. Consequently, the most typical use for such an operation is to move textures from a large collection of system memory surfaces to a surface in video memory in preparation for subsequent operations. System-to-video DMA transfers are about as fast as processor-controlled transfers (for example, HEL blits), but are of great utility since they can operate in parallel with the host processor.

Using DMA

Hardware transfers use physical memory addresses, not the virtual addresses which are home to applications. Some device drivers require that you provide the surface's physical memory address. This mechanism is implemented by the **IDirectDrawSurface3::PageLock** method. If the device driver does not require page locking, the DDCAPS2_NOPAGELOCKREQUIRED flag will be set when you retrieve the hardware capabilities by calling the **IDirectDraw2::GetCaps** method.

Page locking a surface prevents the system from committing a surface's physical memory to other uses, and guarantees that the surface's physical address will remain constant until a corresponding **IDirectDrawSurface3::PageUnlock** call is made. If the device driver requires page locking, DirectDraw will only allow DMA operations on system memory surfaces that the application has page locked. If you do not call **IDirectDrawSurface3::PageLock** in such a situation, DirectDraw will perform the transfers by using software emulation. Note that locking a large amount of system memory will make Windows run poorly. Therefore, it is highly recommended that only full-screen exclusive mode applications use **IDirectDrawSurface3::PageLock** for large amounts of system memory, and that such applications take care to unlock these surfaces when the application is minimized. Of course, when the application is restored, you should page lock the system memory surface again.

Responsibility for managing page locking is entirely in the hands of the application developer. DirectDraw will never page lock or page unlock a surface. Additionally, it is up to you to determine how much memory you can safely page lock without adversely affecting system performance.

Using DirectDraw Palettes in Windowed Mode

IDirectDrawPalette interface methods write directly to the hardware when the display is in exclusive (full-screen) mode. However, when the display is in nonexclusive (windowed) mode, the **IDirectDrawPalette** interface methods call the GDIs palette handling functions to work cooperatively with other windowed applications.

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

- Types of Palette Entries in Windowed Mode
- Creating a Palette in Windowed Mode
- Setting Palette Entries in Windowed Mode

Types of Palette Entries in Windowed Mode

Unlike full-screen exclusive mode applications, windowed applications must share the desktop palette with other applications. This imposes several restrictions on which palette entries you can safely modify and how you can modify them. The **PALETTEENTRY** structure you use when working with DirectDrawPalette objects and GDI contains a **peFlags** member to carry information that describes how the system should interpret the **PALETTEENTRY** structure.

The **peFlags** member describes three types of palette entries, discussed in this topic:

- Windows static entries
- Animated entries
- Nonanimated entries

Windows static entries.

In normal mode, Windows reserves palette entries 0 through 9 and 246 through 255 for system colors that it uses to display menu bars, menu text, window borders, and so on. In order to maintain a consistent look for your application and avoid damaging the appearance of other applications, you need to protect these entries in the palette you set to the primary surface. Often, developers retrieve the system palette entries by calling the **GetSystemPaletteEntries** Win32® function, then explicitly set the identical entries in a custom palette to match before assigning it to the primary surface. Duplicating the system palette entries in a custom palette will work initially, but it becomes invalid if the user changes the desktop color scheme.

To avoid having your palette look bad when the user changes color schemes, you can protect the appropriate entries by providing a reference into the system palette instead specifying a color value. This way, no matter what color the system is using for a given entry, your palette will always match and you won't need to do any updating. The PC_EXPLICIT flag, used in the **peFlags** member, makes it possible for you to directly refer to a system palette entry. When you use this flag, the system no longer assumes that the other structure members include color information. Rather, when you use PC_EXPLICIT, you set the value in the **peRed** member to the desired system palette index and set the other colors to zero.

For instance, if you want to ensure that the proper entries in your palette always match the system's color scheme, you could use the following code:

```
// Set the first and last 10 entries to match the system palette.
PALETTEENTRY pe[256];
ZeroMemory(pe, sizeof(pe));
for(int i=0;i<10;i++){
    pe[i].peFlags  = pe[i+246].peFlags = PC_EXPLICIT;
    pe[i].peRed = i;
    pe[i+246].peRed = i+246;
}
```

You can force Windows to use only the first and last palette entry (0 and 255) by calling the **SetSystemPaletteUse** Win32 function. In this case, you should set only entries 0 and 255 of your **PALETTEENTRY** structure to PC_EXPLICIT.

Animated entries

You specify palette entries that you will be animating by using the PC_RESERVED flag in the corresponding **PALETTEENTRY** structure. Windows will not allow any other application to map its logical palette entry to that physical entry, thereby preventing other applications from cycling their colors when your application animates the palette.

Nonanimated entries

You specify normal, nonanimated palette entries by using the PC_NOCOLLAPSE flag in the corresponding **PALETTEENTRY** structure. The PC_NOCOLLAPSE flag informs Windows not to substitute some other already-allocated physical palette entry for that entry.

Creating a Palette in Windowed Mode

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

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

// First set up the Windows static entries.
for (index = 0; index < 10 ; index++)
{
    // The first 10 static entries:
    pPaletteEntry[index].peFlags = PC_EXPLICIT;
    pPaletteEntry[index].peRed = index;
    pPaletteEntry[index].peGreen = 0;
    pPaletteEntry[index].peBlue = 0;

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

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

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

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

Setting Palette Entries in Windowed Mode

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

In most circumstances, you should not attempt to set any of the Windows static entries when in nonexclusive (windowed) mode or you will get unpredictable results. The only exception is when you reset the 256 entries.

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

The following example illustrates palette animation in nonexclusive mode:

```
LPDIRECTDRAW        lpDD;        // Already initialized
PALETTEENTRY pPaletteEntry[256]; // Already initialized
LPDIRECTDRAWPALETTE lpDDPal;     // Already initialized
int                 index;
HRESULT             ddrval;
PALETTEENTRY        temp;

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

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

Working with Multiple Monitors

Future releases of Windows 95, code named Memphis, and Windows NT support multiple display devices and monitors on a single system. The multiple monitor architecture (casually referred to as “MultiMon”) enables the operating system to use the display area from two or more display devices and monitors to create a single logical desktop. For example, in a MultiMon system with two monitors, the user could display applications on either monitor, or even drag windows from one monitor to another. DirectDraw supports this architecture, but there are a few nuances to be aware of, depending on the cooperative level your application uses.

A DirectDraw application should enumerate the devices, choose a device (or perhaps allow the user to choose the device to use), then create a DirectDraw object for that the device by using its hardware globally unique identifier (GUID). This technique will ensure the best performance on both MultiMon and single monitor systems and at all cooperative levels.

The currently active display device is referred to as the “default device,” or the “null device.” The latter name comes from the fact that the currently active display device is enumerated with NULL as its GUID. Many existing applications create a DirectDraw object for the null device, assuming that the device will be hardware accelerated. However, on multiple monitor systems, the null device isn’t always hardware accelerated; it depends on what cooperative level is set at the time.

In full-screen exclusive mode, the null device is hardware accelerated, but unaware of any other installed devices. This means that full-screen, exclusive mode applications will run as fast on a MultiMon system as any other system, but will not be able to use built-in support for spanning graphics operations across display devices. Full-screen, exclusive mode applications that need to use multiple devices can create a DirectDraw object for each device they want to use. Note that to create a DirectDraw object for a specific device, you must supply that device’s GUID (as it is enumerated when you call **DirectDrawEnumerate**).

When the normal cooperative level is set, the null device has no hardware acceleration; the null device is, effectively, an emulated logical device that combines the resources of two physical devices. Therefore, the null device has no hardware acceleration at all when the normal cooperative level is set. On the other hand, when the normal cooperative level is set, the null device is capable of automatically spanning graphics operations across monitors. As a result, negative coordinates for blit operations are valid when the logical location of secondary monitor is to the left of the primary monitor.

If your application requires hardware acceleration when the normal cooperative level is set, it must create a single DirectDraw object using a specific device’s GUID. Note that when you don’t use the null device, you don’t get automatic device spanning. That is, blit operations that cross an edge of the primary surface will be clipped (if you are using a clipper) or will fail, returning DDERR_INVALIDRECT.

As a rule on any system, you should set the cooperative level immediately after creating a DirectDraw object, before retrieving the object’s capabilities or querying for other interfaces. Additionally, avoid setting the cooperative level multiple times on a MultiMon system. If you need to switch from full-screen to normal mode, it is best to create a new DirectDraw object.

Video Ports

DirectDraw video-port extensions are a low-level programming interface, not intended for mainstream multimedia programmers. The target customer is the video-streaming software industry, which creates products like DirectShow™. Developers who want to include video playback in their software can make use of video-port extensions. However, for most software, a high-level programming interface like the one provided by DirectShow is recommended for greater ease of use.

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

What is a Video-Port Object?

A DirectDrawVideoPort object represents the video-port hardware found on some display adapters. Generally, a video-port object controls how the video-port hardware applies a video signal it receives from a video decoder directly to the frame buffer.

More than one channel of video can be controlled by creating as many DirectDrawVideoPort objects as is required. Because each channel can be separately enumerated and configured, the video hardware for each channel does not need to be identical.

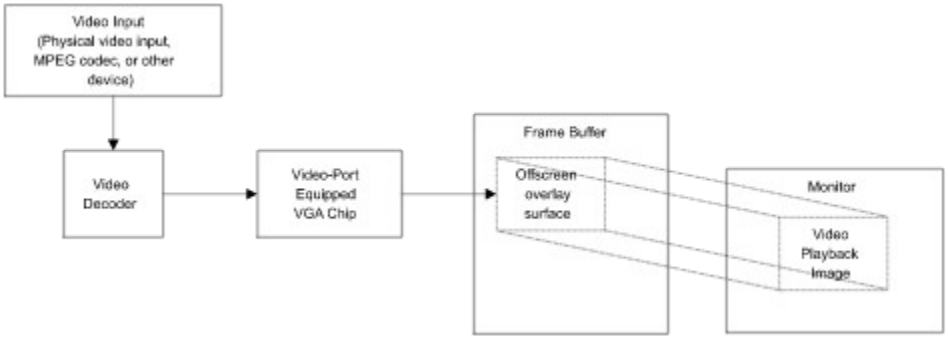
For more information, see Video-Port Technology Overview.

Video-Port Technology Overview

A video port is hardware on a display device that enables direct access to a surface within the frame buffer, bypassing the CPU and PCI bus. Direct frame buffer access makes it possible to efficiently play live or recorded video without creating noticeable load on the CPU. Once in a surface, an image can be displayed on the screen as an overlay, used as a Direct3D texture, or accessed by the CPU for capture or other processing. The following paragraphs provide general information about the components that make up the technology and how they work.

Data Flow

In a machine equipped with a video port, data in a video stream can flow directly from a video source through a video decoder and the video port to the frame buffer. These components often exist together on a display adapter, but can be on separate hardware components that are physically connected to one another. An example of this data flow is provided in the following illustration.



Video source

In the scope of video-port technology, a video source is strictly a hardware video input device, such as a Zoom Video port, MPEG codec, or other hardware source. These sources broadcast signals in a variety of formats, including NTSC, PAL, and SECAM through a physical connection to a video decoder.

Video Decoder

A video decoder is also a hardware component. The video decoder's job is to decipher the information provided by the video source and send it to the video port in an agreed upon connection format. The decoder possesses a physical connection to the video port, and exposes its services through a stream class minidriver. The decoder is responsible for sending video data and clock and sync information to the video port.

Video port

Like the other components in the data flow path, the video port is a piece of hardware. The video port exists on the display adapter's VGA chip and has direct access to the frame buffer. It receives information sent from the decoder, processes it, and places it in the frame buffer to be displayed. During processing, the video port can manipulate image data to provide scaling, shrinking, color control, or cropping services.

Frame Buffer

The frame buffer accepts video data as provided by the video port. Once received, applications can programmatically manipulate the image data, blit it to other locations, or show it on the display using an overlay (the most common function).

About DirectDraw Video-Port Extensions

DirectDraw has been extended to include the DirectDrawVideoPort object, which takes advantage of video-port technology and provides its services through the **IDDVideoPortContainer** and **IDirectDrawVideoPort** interfaces.

DirectDrawVideoPort objects do not control the video decoder, because it provides services of its own, nor does DirectDraw control the video source; it is beyond the scope of the video port. Rather, a DirectDrawVideoPort object represents the video port itself. It monitors the incoming signal and passes image data to the frame buffer, using parameters set though its interface methods to modify the image, perform flipping, or carry out other services.

The **IDDVideoPortContainer** interface, which you can retrieve by calling the **IDirectDraw2::QueryInterface** method, provides methods to query the hardware for its capabilities and create video-port objects. You create a video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method. Video-port objects expose their functionality through the **IDirectDrawVideoPort** interface, enabling you to manipulate the video-port hardware itself. Using these interfaces, you can examine the video-port's capabilities, assign an overlay surface to receive image data, start and stop video playback, and set hardware parameters to manipulate image data for cropping, color control, scaling, or shrinking effects.

DirectDraw video-port extensions provide for multiple video ports on the same machine by allowing you to create multiple DirectDrawVideoPort objects. There is no requirement that multiple video ports on a machine be identical—each port is separately enumerated and configured separately, regardless of any hardware differences that might exist.

In keeping with the general philosophy of DirectX, this technology gives programmers low-level access to hardware features while insulating them from specific hardware implementation details. It is not a high-level API.

Video Frames and Fields

Video can be interlaced or non-interlaced. When a video signal is interlaced, each video frame is made of two fields of image data. Each field is a collection of every other scan line in an image, starting with the first or second scan line. The first field, referred to as the odd field (or field 1), contains the data for the first scan line and skips every other scan line to the end of the image. Similarly, the even field (or field 2), carries every other scan line starting with the second. The “even-ness” or “odd-ness” of a field is referred to as its field polarity.

When video is not interlaced, each field contains all of a frame's scan lines. Typically, video signals are sent at a rate of 30 frames per second; in the case of interleaved video, this means the rate is 60 fields per second.

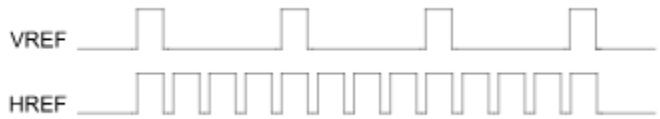
The fields that make up a frame do not always reflect the same moment in time. For example, if the frames are separated by 1/30 of a second then the two fields of a frame may be separated by 1/60 of a second. Because a television displays each field individually, no two fields are simultaneously visible, and the difference between fields adds to the illusion of movement.

HREF, VREF, and Connections

When a monitor or other display device is displaying an image, it typically scans down the screen, creating an image from left to right, top to bottom. (Sometimes, the device makes two passes down the screen to create a single image; this type of display is called an interlaced display.) The video stream contains signals that instruct the display device when a new line or new screen is to be drawn.

The terms HREF and VREF, also known as hsync and vsync, are the signals within the video stream that tell a display device what to do and when to do it. The HREF signals that a new line is to be drawn and the VREF signals a new screen.

For instance, imagine you're working with a video signal intended for the world's smallest monitor. The monitor only has 4 scan lines. (This is not at all realistic, of course, but it's simple.) On an oscilloscope, the HREF and VREF signals would look somewhat like the following illustration:



In the preceding illustration, both HREF and VREF signals are "active high," meaning that they are considered active when in a heightened state (when the waves go up). There is no standard for these signals. In some cases, places where the waves go down ("low" states) might signal an active HREF or VREF, or sometimes one will be active high and the other active low. Although the preceding illustration is only an imaginary example, note that there are lots of HREF signals for each VREF. This is because for each new screen, there are several scan lines. Of course, in a real video signal for a real broadcast, you would see hundreds of HREFs for a single VREF.

HREF signals, VREF signals, and video data are carried across physical data lines from the decoder to the video port. In many cases, a number of lines are reserved for video data, and others are dedicated to carrying HREF and VREF signals. However, there is no standard for how these data lines are used.

A connection is a protocol that a video port or decoder uses to define how it uses these data lines. Video ports and video decoders will support a variety of connections. DirectDraw video-port extensions use globally-unique identifiers (GUIDs) to identify each type of connection. You can query for the connections that the video port supports by calling the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. You create a DirectDrawVideoPort object that supports a given connection by calling the **IDDVideoPortContainer::CreateVideoPort** method.

Keep in mind that the video decoder is outside the scope of DirectDraw video-port extensions, and exposes its supported connections through an interface of its own. By enumerating the connections that the video-port supports and comparing the results with the connections supported by the decoder, you can negotiate a common connection (or "language") that both components understand.

Vertical Blanking Interval Data

In broadcast video, a small period of time elapses between video frames, during which a display device refreshes its display for the next frame. This period of time is called the Vertical Blanking Interval (VBI). Instead of sitting idle during the VBI, broadcast video encodes data in the first twenty-one scan lines of a video frame and sends these lines during the VBI. This data is often used for closed captioning or time-stamping, but can be used for other purposes.

DirectDraw video-port extensions enable you to divert data contained with the VBI to a surface, bypass scaling of VBI data, and automatically flip between VBI surfaces in a flipping chain. Once data is in a surface, you can directly access the surface's memory as needed.

For more information, see Auto-flipping.

Auto-flipping

To avoid tearing images when refreshing the screen between frames, DirectDrawVideoPort objects can automatically flip their target overlay surfaces in response to VREF signals. To use this service, the target surface you set to the video-port object with the **IDirectDrawVideoPort::SetTargetSurface** method must be the first surface in a flipping chain of overlay surfaces. Then, to begin playing the video sequence, call the **IDirectDrawVideoPort::StartVideo** method, specifying the DDVP_AUTOFLIP flag in the **dwVPFlags** member of the associated **DDVIDEOPORTINFO** structure. The video-port object will flip to the next surface in the flipping chain for each VREF signal is receives. If the video port is interleaving fields, it will flip once for every two VREF signals it receives.

If you are using auto-flipping and want to direct VBI data to separate auto-flipped surfaces, you must have the same number of VBI surfaces as you do standard video surfaces.

Solutions to Common Video Artifacts

Several problems are inherent in displaying broadcast video on display devices other than televisions. This section briefly discusses some common problems, then describes how DirectDraw video-port extensions tries to solve them.

NTSC Interlaced Display and Interleaved Memory

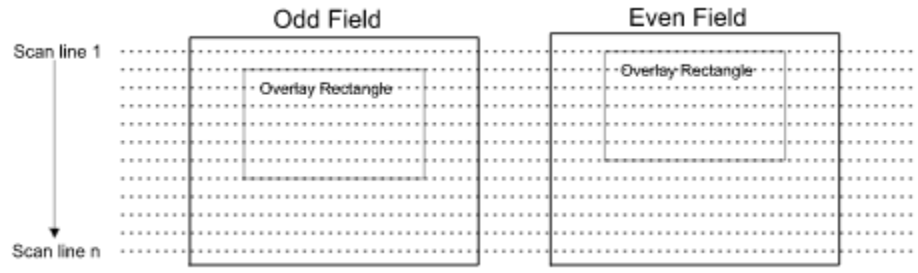
An NTSC signal broadcasts video at an approximate rate of 30 frames, or 60 fields, per second. Like a frame, a field in an NTSC signal is independent of the other field in a frame and can contain different image data. For more information on this behavior, see Video Frames and Fields.

The problems caused by the independence of fields within a frame become apparent when two fields are interleaved for display. In video with a lot of movement, the two fields of a single frame will contain images that don't match each other, resulting in motion artifacts.

One way that developers have tried to work around this behavior is by discarding one of the fields. This solution causes a loss in image quality by roughly one-half, but provides acceptable results for some purposes. Another method frequently used is to display fields individually, stretching each vertically by a factor of two when it is displayed. This provides better image quality, but because fields are offset by one pixel in the Y direction, the result is an animation that "jitters" up and down as it plays.

DirectDraw video-port extensions can employ two, more advanced, techniques for improving image quality, known as "Bob" and "Weave." Both are supported by the DirectDraw overlay surfaces that are used with video-port extensions.

The first algorithm, "Bob," is very similar to the method of displaying each field in a frame individually. However, for each field, the overlay's source rectangle is adjusted to accommodate for any jittering effects. Effectively, the source rectangle bounces up and down in time with the fields, negating the jittering onscreen. The following illustration depicts this process.



The "Weave" algorithm provides the best image quality for material that originates from film by exploiting a common technique used in the video industry for converting motion pictures to television. Unlike Bob, a video-port object does not Weave by itself; you must combine the default overlay behavior of displaying both fields simultaneously with kernel mode video transport (to be provided a future release of Windows 95, code named Memphis, and Windows NT) to implement the algorithm.

Here is a synopsis of the algorithm, provided for completeness. Motion pictures capture video at a rate of 24 frames per second. When converting a motion picture for television, technicians use a technique called "3:2 pulldown" to convert the frame rate to the 30 frames per second required for television broadcasts. This technique involves inserting a redundant field for every four true fields in the video stream to come up with the required number of fields.

When you "weave," you are reversing this process. You detect when 3:2 pulldown is being used, removing any redundant fields to restore the original motion-picture frames. The fields that make up the restored frames can then be interleaved in memory without risk of motion artifacts. Occasionally, the pattern of redundant frames will change due to edits within the original film or reel breaks. You must monitor when these changes occur and update the behavior to adjust for the new pattern.

By default, an overlay surface displays both fields simultaneously. This works well if you're implementing the Weave algorithm, but prevents the video port from using the Bob algorithm. You can

programmatically change how the overlay treats video data by calling the **IDirectDrawSurface3::UpdateOverlay** method. The flags you include in the **dwFlags** parameter determine the overlay's behavior: if you include the DDOVER_BOB flag, the video port will use the Bobbing algorithm; if you don't, it displays both fields. Note that by simply displaying both fields simultaneously, the resulting video will show motion artifacts.

Solving Problems Caused by Half-Lines

Some video decoders output a half line of meaningless data at the beginning of the even field. If this extra line is written to the frame buffer, the resulting image will appear garbled. In some cases, the video-port hardware is capable of sensing and discarding this data before writing it to the frame buffer.

You can determine if a video port is capable of discarding this data when retrieving connection information with the **IDDVideoPortContainer::GetVideoPortConnectInfo** method. If the video port cannot discard half-lines, the DDVPCONNECT_HALFLINE flag will be specified in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure for each supported connection.

If the video port is unable to discard half-lines, you have two options: you can discard one of the fields, or you can work around the hardware's limitations by making some adjustments in how you create the video-port object, and display images with the target overlay surface

Here's how to work around the problem. When creating the video-port object by calling the **IDDVideoPortContainer::CreateVideoPort** method, include the DDVPCONNECT_INVERTPOLARITY flag in the **dwFlags** member of the associated **DDVIDEOPORTCONNECT** structure. This causes the video port to invert the polarity of the fields in the video stream, treating even fields like odd fields and vice versa. Once reversed, the half-line preceding even fields will be written to the frame buffer as the first scan line of each frame. To remove the unwanted data, adjust the source rectangle of the overlay surface used to display the image down one pixel by calling the **IDirectDrawVideoPort::StartVideo** method with the necessary coordinates. Note that this technique requires that you allocate one extra line in the surface containing the even field.

Exploiting Hardware Features

Video-port hardware often supports special features for adjusting color, shrinking or zooming images, handling VBI data, or skipping fields. The HAL provides information about these features by using flags in the **DDVIDEOPORTCAPS** structure. You retrieve the capabilities of a machine's video-port hardware by calling the **IDDVideoPortContainer::EnumVideoPorts** method.

To exploit these features for playback, you use the **IDirectDrawVideoPort::StartVideo** method, which uses a **DDVIDEOPORTINFO** structure to request that hardware features be used to modify image data before placing it in the frame buffer or for display. By setting values and flags in this structure, you can specify the source rectangle used with the overlay surface, indicate cropping regions, request hardware scaling, and set pixel formats.

DirectDrawVideoPort objects do not emulate video-port hardware services.

Getting the Flip and Blit Status

When the **IDirectDrawSurface3::Flip** method is called, the primary surface and back buffer are exchanged. However, the exchange may not occur immediately. For example, if a previous flip has not finished, or if it did not succeed, this method returns **DDERR_WASSTILLDRAWING**. In the samples included with this SDK, the **IDirectDrawSurface3::Flip** call continues to loop until it returns DD_OK. Also, a **IDirectDrawSurface3::Flip** call does not complete immediately. It schedules a flip for the next time a vertical blank occurs on the system.

An application that waits until the **DDERR_WASSTILLDRAWING** value is not returned is very inefficient. Instead, you could create a function in your application that calls the **IDirectDrawSurface3::GetFlipStatus** method on the back buffer to determine if the previous flip has finished.

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

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

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

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

You can use the **IDirectDrawSurface3::GetBltStatus** method in much the same way to determine whether a blit has finished. Because **IDirectDrawSurface3::GetFlipStatus** and **IDirectDrawSurface3::GetBltStatus** return immediately, you can use them periodically in your application with little loss in speed.

Blitting with Color Fill

You can use the **IDirectDrawSurface3::Blt** method to perform a color fill of the most common color you want to be displayed. For example, if the most common color your application displays is blue, you can use **IDirectDrawSurface3::Blt** with the DDBLT_COLORFILL flag to first fill the surface with the color blue. Then you can write everything else on top of it. This allows you to fill in the most common color very quickly, and you then only have to write a minimum number of colors to the surface.

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

```
DDBLTFX ddbltfx;

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

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

Determining the Capabilities of the Display Hardware

DirectDraw uses software emulation to perform the DirectDraw functions not supported by the user's hardware. To accelerate performance of your DirectDraw applications, you should determine the capabilities of the user's display hardware after you have created a DirectDraw object, then structure your program to take advantage of these capabilities when possible.

You can determine these capabilities by using the **IDirectDraw2::GetCaps** method. Not all hardware features are supported in emulation. If you want to use a feature only supported by some hardware, you must also be prepared to supply some alternative for systems with hardware that lacks that feature.

Storing Bitmaps in Display Memory

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

Most display adapter hardware contains enough extra memory to store more than only the primary surface and the back buffer. You can use the **dwVidMemTotal** and **dwVidMemFree** members of the **DDCAPS** structure (if you used the **IDirectDraw2::GetCaps** method to get the capabilities of the user's display hardware) to determine the amount of available memory for storing bitmaps in the display adapter's memory. If you want to see how this works, use the DirectX Viewer sample application included with the DirectX APIs in the Platform SDK. Under DirectDraw Devices, open the Primary Display Driver folder, and then open the General folder. The amount of total display memory (minus the primary surface) and the amount of free memory is displayed. Each time a surface is added to the DirectDraw object, the amount of free memory decreases by the amount of memory used by the added surface.

Triple Buffering

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

```
// The lpDDSPrimary, lpDDSMiddle, and lpDDSBack are globally
// declared, uninitialized LPDIRECTDRAWSURFACE variables.

DDSURFACEDESC ddsd;
ZeroMemory (&ddsd, sizeof(ddsd));

// Create the primary surface with two back buffers.
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
    DDSCAPS_FLIP | DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 2;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);

// If we successfully created the flipping chain,
// retrieve pointers to the surfaces we need for
// flipping and blitting.
if(ddrval == DD_OK)
{
    // Get the surface directly attached to the primary (the back buffer).
    ddsd.ddsCaps.dwCaps = DDSCAPS_BACKBUFFER;
    ddrval = lpDDSPrimary->GetAttachedSurface(&ddsd.ddsCaps,
        &lpDDSMiddle);
    if(ddrval != DD_OK) ;
        // Display an error message here.
}
```

You do not need to keep track of all surfaces in a triple buffered flipping chain. The only surfaces you must keep pointers to are the primary surface and the back-buffer surface. You need a pointer to the primary surface in order to flip the surfaces in the flipping chain, and you need a pointer to the back buffer for blitting. For more information, see Flipping Surfaces.

Triple buffering allows your application to continue blitting to the back buffer even if a flip has not completed and the back buffer's blit has already finished. Performing a flip is not a synchronous event; one flip can take longer than another. Therefore, if your application uses only one back buffer, it may spend some time idling while waiting for the **IDirectDrawSurface3::Flip** method to return with DD_OK.

DirectDraw Applications and Window Styles

If your application uses DirectDraw in windowed mode, you can create windows with any window style. However, full screen, exclusive mode applications cannot be created with the WS_EX_TOOLWINDOW style without risk of unpredictable behavior. The WS_EX_TOOLWINDOW style prevents a window from being the top most window, which is required for a DirectDraw full screen, exclusive mode application.

Full screen exclusive mode applications should use the WS_EX_TOPMOST extended window style and the WS_VISIBLE window style to display properly. These styles keep the application at the front of the window z-order and prevent GDI from drawing on the primary surface.

The following example shows one way to safely prepare a window to be used in a full-screen, exclusive mode application.

```
//////////////////////////////////////////////////////
// Register the window class, display the window, and init
// all DirectX and graphic objects.
//////////////////////////////////////////////////////
BOOL WINAPI InitApp(INT nWinMode)
{
    WNDCLASSEX wcex;

    wcex.cbSize           =    sizeof(WNDCLASSEX);
    wcex.hInstance        =    g_hinst;
    wcex.lpszClassName    =    g_szWinName;
    wcex.lpfnWndProc      =    WndProc;
    wcex.style            =    CS_VREDRAW|CS_HREDRAW|CS_DBLCLKS;
    wcex.hIcon            =    LoadIcon (NULL, IDI_APPLICATION);
    wcex.hIconSm          =    LoadIcon (NULL, IDI_WINLOGO);
    wcex.hCursor          =    LoadCursor (NULL, IDC_ARROW);
    wcex.lpszMenuName     =    MAKEINTRESOURCE(IDR_APPMENU);
    wcex.cbClsExtra       =    0 ;
    wcex.cbWndExtra       =    0 ;
    wcex.hbrBackground    =    GetStockObject (NULL_BRUSH);

    RegisterClassEx(&wcex);

    g_hwndMain = CreateWindowEx(
                        WS_EX_TOPMOST,
                        g_szWinName,
                        g_szWinCaption,
                        WS_VISIBLE|WS_POPUP,
                        0,0,CX_SCREEN,CY_SCREEN,
                        NULL,
                        NULL,
                        g_hinst,
                        NULL);

    if(!g_hwndMain)
        return(FALSE);

    SetFocus(g_hwndMain);
    ShowWindow(g_hwndMain, nWinMode);
    UpdateWindow(g_hwndMain);

    return TRUE;
```

```
}
```

Matching True RGB Colors to the Frame Buffer's Color Space

Applications often need to find out how a true RGB color (RGB 888) will be mapped into a the frame buffer's color space when the display device is not in RGB 888 mode. For example, imagine you're working on an application that will run in 16 and 24 bit RGB display modes. You know that when the art was created, a color was reserved for use as a transparent blitting color key; for the sake of argument, it is a 24 bit color such as RGB(128,64,255). Because your application will also run in a 16 bit RGB mode, you need a way to find out how this 24 bit color key maps into the color space that the frame buffer uses when it's running in a 16 bit RGB mode.

Although DirectDraw does not perform color matching services for you, there are ways to calculate how your color key will be mapped in the frame buffer. These methods can be pretty complicated. For most purposes, you can use the GDI built-in color matching services, combined with the DirectDraw direct frame buffer access, to determine how a color value maps into a different color space. In fact, the Ddutil.cpp source file included in the DirectX examples of the Platform SDK includes a sample function called **DDColorMatch** that performs this task. The **DDColorMatch** sample function performs the following main tasks:

1. Retrieves the color value of a pixel in a surface at 0,0.
2. Calls the Win32 **SetPixel** function, using a **COLORREF** structure that describes your 24-bit RGB color.
3. Uses DirectDraw to lock the surface, getting a pointer to the frame buffer memory.
4. Retrieves the actual color value from the frame buffer (set by GDI in Step 2) and unlocks the surface
5. Resets the pixel at 0,0 to its original color using **SetPixel**.

The process used by the **DDColorMatch** sample function is not fast; it isn't intended to be. However, it provides a reliable way to determine how a color will be mapped across different RGB color spaces. For more information, see the source code for **DDColorMatch** in the Ddutil.cpp source file.

DirectDraw Tutorials

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

- Tutorial 1: The Basics of DirectDraw
- Tutorial 2: Loading Bitmaps on the Back Buffer
- Tutorial 3: Blitting from an Off-Screen Surface
- Tutorial 4: Color Keys and Bitmap Animation
- Tutorial 5: Dynamically Modifying Palettes
- Tutorial 6: Using Overlay Surfaces

Some samples in these tutorials use the older **IDirectDraw** and **IDirectDrawSurface** interfaces. If you want to update these examples so they use the DirectX 5 interfaces query for the new versions of the interfaces before using them. In addition, you must change the appropriate parameters of any methods that have been updated for new versions of the interfaces.

Note The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you need to add the *vtable* and *this* pointers to the interface methods.

Tutorial 1: The Basics of DirectDraw

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

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

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

Note: To use GUIDs successfully in your applications, you must either define INITGUID prior to all other include and define statements, or you must link to the DXGUID.LIB library. You should define INITGUID in only one of your source modules.

Step 1: Creating a DirectDraw Object

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

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

```
ddrval = DirectDrawCreate(NULL, &lpDD, NULL);
if(ddrval == DD_OK)
{
    // lpDD is a valid DirectDraw object.
}
else
{
    // The DirectDraw object could not be created.
}
```

Step 2: Determining the Application's Behavior

Before you can change the resolution of your display, you must at a minimum specify the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags in the *dwFlags* parameter of the **IDirectDraw2::SetCooperativeLevel** method. This gives your application complete control over the display device, and no other application will be able to share it. In addition, the DDSCL_FULLSCREEN flag sets the application in exclusive (full-screen) mode. Your application covers the entire desktop, and only your application can write to the screen. The desktop is still available, however. (To see the desktop in an application running in exclusive mode, start DDEX1 and press ALT + TAB.)

The following example demonstrates the use of the **IDirectDraw2::SetCooperativeLevel** method:

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

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

If **IDirectDraw2::SetCooperativeLevel** does not return DD_OK, you can still run your application. The application will not be in exclusive mode, however, and it might not be capable of the performance your application requires. In this case, you might want to display a message that allows the user to decide whether or not to continue.

One requirement for using **IDirectDraw2::SetCooperativeLevel** is that you must pass a handle of a window (**HWND**) to allow Windows to determine if your application terminates abnormally. For example, if a general protection (GP) fault occurs and GDI is flipped to the back buffer, the user will not be able to return to the Windows screen. To prevent this from occurring, DirectDraw provides a process running in the background that traps messages that are sent to that window. DirectDraw uses these messages to determine when the application terminates. This feature imposes some restrictions, however. You have to specify the window handle that is retrieving messages for your application—that is, if you create another window, you must ensure that you specify the window that is active. Otherwise, you might experience problems, including unpredictable behavior from GDI, or no response when you press ALT+TAB.

Step 3: Changing the Display Mode

After you have set the application's behavior, you can use the **IDirectDraw2::SetDisplayMode** method to change the resolution of the display. The following example shows how to set the display mode to 640×480×8 bpp:

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

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

When you set the display mode, you should ensure that if the user's hardware cannot support higher resolutions, your application reverts to a standard mode that is supported by a majority of display adapters. For example, your application could be designed to run on all systems that support 640×480×8 as a standard backup resolution.

Note: **IDirectDraw2::SetDisplayMode** returns a DDERR_INVALIDMODE error value if the display adapter could not be set to the desired resolution. Therefore, you should use the **IDirectDraw2::EnumDisplayModes** method to determine the capabilities of the user's display adapter before trying to set the display mode.

Step 4: Creating Flipping Surfaces

After you have set the display mode, you must create the surfaces on which to place your application. Because the DDEX1 example is using the **IDirectDraw2::SetCooperativeLevel** method to set the mode to exclusive (full-screen) mode, you can create surfaces that flip between the surfaces. If you were using **IDirectDraw2::SetCooperativeLevel** to set the mode to DDSCL_NORMAL, you could create only surfaces that blit between the surfaces. Creating flipping surfaces requires the following steps, also discussed in this topic:

- Defining the surface requirements
- Creating the surfaces

Defining the Surface Requirements

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

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

ddsd.dwBackBufferCount = 1;
```

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

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

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

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

Surface memory can be either display memory or system memory. DirectDraw uses system memory if the application runs out of display memory (for example, if you specify more than one back buffer on a display adapter with only 1 MB of RAM). You can also specify whether to use only system memory or only display memory by setting the **dwCaps** member in the **DDSCAPS** structure to DDSCAPS_SYSTEMMEMORY or DDSCAPS_VIDEOMEMORY. (If you specify DDSCAPS_VIDEOMEMORY, but not enough memory is available to create the surface, **IDirectDraw2::CreateSurface** returns with a DDERR_OUTOFVIDEOMEMORY error.)

Creating the Surfaces

After the **DDSURFACEDESC** structure is filled, you can use it and *lpDD*, the pointer to the DirectDraw object that was created by the **DirectDrawCreate** function, to call the **IDirectDraw2::CreateSurface** method, as shown in the following example:

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

The *lpDDSPrimary* parameter will point to the primary surface returned by **IDirectDraw2::CreateSurface** if the call succeeds.

After the pointer to the primary surface is available, you can use the **IDirectDrawSurface3::GetAttachedSurface** method to retrieve a pointer to the back buffer, as shown in the following example:

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

By supplying the address of the surface's primary surface and by setting the capabilities value with the DDSCAPS_BACKBUFFER flag, the *lpDDSBack* parameter will point to the back buffer if the **IDirectDrawSurface3::GetAttachedSurface** call succeeds.

Step 5: Rendering to the Surfaces

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

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

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

The example uses the **IDirectDrawSurface3::GetDC** method to retrieve the handle of the device context, and it internally locks the surface. If you are not going to use Windows functions that require a handle of a device context, you could use the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods to lock and unlock the back buffer.

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

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

After the text has been written to the buffer, the example uses the **IDirectDrawSurface3::ReleaseDC** method to unlock the surface and release the handle. Whenever your application finishes writing to the back buffer, you must call either **IDirectDrawSurface3::ReleaseDC** or **IDirectDrawSurface3::Unlock**, depending on your application. Your application cannot flip the surface until the surface is unlocked.

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

Note After the surface is unlocked by using **IDirectDrawSurface3::Unlock**, the pointer to the surface memory is invalid. You must use **IDirectDrawSurface3::Lock** again to obtain a valid pointer to the surface memory.

Step 6: Writing to the Surface

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

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

The line of code that calls the **IDirectDrawSurface3::GetDC** method locks the back buffer in preparation for writing. The **SetBkColor** and **SetTextColor** functions set the colors of the background and text.

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

After the message is written to the back buffer, the back buffer is unlocked by using the **IDirectDrawSurface3::ReleaseDC** method.

Step 7: Flipping the Surfaces

After the surface memory is unlocked, you can use the **IDirectDrawSurface3::Flip** method to flip the back buffer to the primary surface, as shown in the following example:

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

In the example, **lpDDSPrimary** designates the primary surface and its associated back buffer. When **IDirectDrawSurface3::Flip** is called, the front and back surfaces are exchanged (only the pointers to the surfaces are changed; no data is actually moved). If the flip is successful and returns DD_OK, the application breaks from the while loop.

If the flip returns with a **DDERR_SURFACELOST** value, an attempt is made to restore the surface by using the **IDirectDrawSurface3::Restore** method. If the restore is successful, the application loops back to the **IDirectDrawSurface3::Flip** call and tries again. If the restore is unsuccessful, the application breaks from the while loop, and returns with an error.

Note When you call **IDirectDrawSurface3::Flip**, the flip does not complete immediately. Rather, a flip is scheduled for the next time a vertical blank occurs on the system. If, for example, the previous flip has not occurred, **IDirectDrawSurface3::Flip** returns **DDERR_WASSTILLDRAWING**. In the example, the **IDirectDrawSurface3::Flip** call continues to loop until it returns DD_OK.

Step 8: Deallocating the DirectDraw Objects

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

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

The application checks if the pointers to the DirectDraw object (*lpDD*) and the DirectDrawSurface object (*lpDDSPrimary*) are not equal to NULL. Then DDEX1 calls the **IDirectDrawSurface3::Release** method to decrease the reference count of the DirectDrawSurface object by 1. Because this brings the reference count to 0, the DirectDrawSurface object is deallocated. The DirectDrawSurface pointer is then destroyed by setting its value to NULL. Next, the application calls **IDirectDraw2::Release** to decrease the reference count of the DirectDraw object to 0, deallocating the DirectDraw object. This pointer is then also destroyed by setting its value to NULL.

Tutorial 2: Loading Bitmaps on the Back Buffer

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

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

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

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

if (lpDDPal == NULL)
    goto error;

ddrval = lpDDSPrimary->SetPalette(lpDDPal);

if(ddrval != DD_OK)
    goto error;

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

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

Step 1: Creating the Palette

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

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

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

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

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

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

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

When the **IDirectDraw2::CreatePalette** method returns, the *ddpal* parameter points to the DirectDrawPalette object, which is then returned from the **DDLoadPalette** call.

The *ape* parameter is a pointer to a structure that can contain either 2, 4, 16, or 256 entries, organized linearly. The number of entries depends on the *dwFlags* parameter in the **IDirectDraw2::CreatePalette** method. In this case, the *dwFlags* parameter is set to DDPCAPS_8BIT, which indicates that there are 256 entries in this structure. Each entry contains 4 bytes (a red channel, a green channel, a blue channel, and a flags byte).

Step 2: Setting the Palette

After you create the palette, you pass the pointer to the DirectDrawPalette object (*ddpal*) to the primary surface by calling the **IDirectDrawSurface3::SetPalette** method, as shown in the following example:

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

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

After you have called **IDirectDrawSurface3::SetPalette**, the DirectDrawPalette object is associated with the DirectDrawSurface object. Any time you need to change the palette, you simply create a new palette and set the palette again. (Although this tutorial uses these steps, there are other ways of changing the palette, as will be shown in later examples.)

Step 3: Loading a Bitmap on the Back Buffer

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

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

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

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

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

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

The *ddsd* value is a pointer to the **DDSURFACEDESC** structure. This structure stores the current description of the DirectDraw surface. In this case, the **DDSURFACEDESC** members describe the height and width of the surface, which are indicated by DDSD_HEIGHT and DDSD_WIDTH. The call to the **IDirectDrawSurface3::GetSurfaceDesc** method then loads the structure with the proper values. For DDEX2, the values will be 480 for the height and 640 for the width.

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

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

Step 4: Flipping the Surfaces

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

Tutorial 3: Blitting from an Off-Screen Surface

The sample in Tutorial 2 (DDEX2) takes a bitmap and puts it in the back buffer, and then it flips between the back buffer and the primary buffer. This is not a very realistic approach to displaying bitmaps. The sample in this tutorial (DDEX3) expands on the capabilities of DDEX2 by including two off-screen buffers in which the two bitmaps—one for the even screen and one for the odd screen—are stored. It uses the **IDirectDrawSurface3::BltFast** method to copy the contents of an off-screen surface to the back buffer, and then it flips the buffers and copies the next off-screen surface to the back buffer.

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

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

Step 1: Creating the Off-Screen Surfaces

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

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

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

The **dwFlags** member specifies that the application will use the **DDSCAPS** structure, and it will set the height and width of the buffer. The surface will be an off-screen plain buffer, as indicated by the DDSCAPS_OFFSCREEN flag set in the **DDSCAPS** structure. The height and the width are set as 480 and 640, respectively, in the **DDSURFACEDESC** structure. The surface is then created by using the **IDirectDraw2::CreateSurface** method.

Because both of the off-screen plain buffers are the same size, the only requirement for creating the second buffer is to call **IDirectDraw2::CreateSurface** again with a different pointer name.

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

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

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

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

if (hbm == NULL)
    return FALSE;

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

return TRUE;
```

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

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

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

```
rcRect.left = 0;
rcRect.top = 0;
rcRect.right = 640;
rcRect.bottom = 480;
if(phase)
{
    pdds = lpDDSTwo;
    phase = 0;
}
else
{
    pdds = lpDDSOne;
    phase = 1;
}
while(1)
{
    ddrval = lpDDSBack->BltFast(0, 0, pdds, &rcRect, FALSE);
    if(ddrval == DD_OK)
    {
        break;
    }
```

The phase variable determines which off-screen surface will be blitted to the back buffer. The **IDirectDrawSurface3::BltFast** method is then called to blit the selected off-screen surface onto the back buffer, starting at position (0, 0), the upper-left corner. The *rcRect* parameter points to the **RECT** structure that defines the upper-left and lower-right corners of the off-screen surface that will be blitted from. The last parameter is set to FALSE (or 0), indicating that no specific transfer flags are used.

Depending on the requirements of your application, you could use either the **IDirectDrawSurface3::Blt** method or the **IDirectDrawSurface::BltFast** method to blit from the off-screen buffer. If you are performing a blit from an off-screen plain buffer that is in display memory, you should use **IDirectDrawSurface3::BltFast**. Although you will not gain speed on systems that use hardware blitter on their display adapters, the blit will take about 10 percent less time on systems that use hardware emulation to perform the blit. Because of this, you should use **IDirectDrawSurface3::BltFast** for all display operations that blit from display memory to display memory. If you are blitting from system memory or require special hardware flags, however, you have to use **IDirectDrawSurface3::Blt**.

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

Tutorial 4: Color Keys and Bitmap Animation

The sample in Tutorial 3 (DDEX3) shows one simple method of placing bitmaps into an off-screen buffer before they are blitted to the back buffer. The sample in this tutorial (DDEX4) uses the techniques described in the previous tutorials to load a background and a series of sprites into an off-screen surface. Then it uses the **IDirectDrawSurface3::BltFast** method to copy portions of the off-screen surface to the back buffer, thereby generating a simple bitmap animation.

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

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

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

Step 1: Setting the Color Key

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

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

return TRUE;
```

You can select the color key by setting the RGB values for the color you want in the call to the **DDSetColorKey** function. The RGB value for black is (0, 0, 0). The **DDSetColorKey** function calls the **DDColorMatch** function. (Both functions are in Ddutil.cpp.) The **DDColorMatch** function stores the current color value of the pixel at location (0, 0) on the bitmap located in the *lpDDSOne* surface. Then it takes the RGB values you supplied and sets the pixel at location (0, 0) to that color. Finally, it masks the value of the color with the number of bits per pixel that are available. After that is done, the original color is put back in location (0, 0), and the call returns to **DDSetColorKey** with the actual color key value. After it is returned, the color key value is placed in the **dwColorSpaceLowValue** member of the **DDCOLORKEY** structure. It is also copied to the **dwColorSpaceHighValue** member. The call to **IDirectDrawSurface3::SetColorKey** then sets the color key.

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

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

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

Step 2: Creating a Simple Animation

The DDEX4 sample uses the **updateFrame** function to create a simple animation using the red donuts included in the All.bmp file. The animation consists of three red donuts positioned in a triangle and rotating at various speeds. This sample compares the Win32 **GetTickCount** function with the number of milliseconds since the last call to **GetTickCount** to determine whether to redraw any of the sprites. It subsequently uses the **IDirectDrawSurface3::BltFast** method first to blit the background from the off-screen surface (*lpDDSOne*) to the back buffer, and then to blit the sprites to the back buffer using the color key that you set earlier to determine which pixels are transparent. After the sprites are blitted to the back buffer, DDEX4 calls the **IDirectDrawSurface3::Flip** method to flip the back buffer and the primary surface.

Note that when you use **IDirectDrawSurface3::BltFast** to blit the background from the off-screen surface, the *dwTrans* parameter that specifies the type of transfer is set to DDBLTFAST_NOCOLORKEY. This indicates that a normal blit will occur with no transparency bits. Later, when the red donuts are blitted to the back buffer, the *dwTrans* parameter is set to DDBLTFAST_SRCCOLORKEY. This indicates that a blit will occur with the color key for transparency as it is defined, in this case, in the *lpDDSOne* buffer.

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

Tutorial 5: Dynamically Modifying Palettes

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

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

Step 1: Loading the Palette Entries

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

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

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

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

lpDDSOne->Unlock(NULL);
```

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

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

The color index values that are set in **torusColors** will be used later to determine which colors in the palette are rotated. Because there are no common colors between the background and the red donuts, only those colors associated with the red donuts are rotated. If you want to check whether this is true or not, just remove the "*ddsd.lPitch" from the array and see what happens when you recompile and run the program. (Without multiplying y×**lPitch**, the red donuts are never reached and only the colors found in the background are indexed and later rotated.) For more information about width and pitch, see Width and Pitch.

Step 2: Rotating the Palettes

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

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

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

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

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

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

With this done, the surfaces are subsequently flipped.

Tutorial 6: Using Overlay Surfaces

This tutorial shows you, step by step, how to use DirectDraw and hardware supported overlay surfaces in your applications. The tutorial is written around the Mosquito sample application included with the DirectX SDK samples. The Mosquito sample is a simple application that uses a flipping chain of overlay surfaces to display an animated bitmap on the desktop without blitting to the primary surface. The sample adjusts the characteristics of the overlay surface as needed to accommodate for hardware limitations.

The Mosquito sample application performs the following steps (complex tasks are divided into smaller sub-steps):

Step 1: Creating a Primary Surface

To prepare for using overlay surfaces, you must first initialize DirectDraw and create a primary surface over which the overlay surface will be displayed. Mosquito creates a primary surface with the following code:

```
// Zero-out the structure and set the dwSize member.
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);

// Set flags and create a primary surface.
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
ddrval = g_lpdd->CreateSurface(&ddsd, &g_lpddsPrimary, NULL );
```

The preceding example begins by initializing the **DDSURFACEDESC** structure it will use. It then sets the flags appropriate to create a primary surface and creates it by calling the **IDirectDraw2::CreateSurface** method. For the call, the first parameter is a pointer to a **DDSURFACEDESC** structure that describes the surface to be created. The second parameter is a pointer to a variable that will receive an **IDirectDrawSurface** interface pointer if the call succeeds. The last parameter is set to NULL to indicate that no COM aggregation is taking place.

Step 2: Testing for Hardware Overlay Support

After initializing DirectDraw, you need to verify that the device supports overlay surfaces. Because DirectDraw doesn't emulate overlays, if the hardware device driver doesn't support them, you can't continue. You can test for overlay support by retrieving the device driver capabilities with the **IDirectDraw2::GetCaps** method. After the call, look for the presence of the DDCAPS_OVERLAY flag in the **dwFlags** member of the associated **DDCAPS** structure. If the flag is present, then the display hardware supports overlays; if not, you can't use overlay surfaces with that device.

The following example, taken from the Mosquito sample application, shows how to test for hardware overlay support.

```
BOOL AreOverlaysSupported()
{
    DDCAPS  capsDrv;
    HRESULT ddrval;

    // Get driver capabilities to determine Overlay support.
    ZeroMemory(&capsDrv, sizeof(capsDrv));
    capsDrv.dwSize = sizeof(capsDrv);

    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
        return FALSE;

    // Does the driver support overlays in the current mode?
    // (Currently the DirectDraw emulation layer does not support overlays.
    // Overlay related APIs will fail without hardware support).
    if (!(capsDrv.dwCaps & DDCAPS_OVERLAY))
        return FALSE;

    return TRUE;
}
```

The preceding example calls the **IDirectDraw2::GetCaps** method to retrieve device driver capabilities. The first parameter for the call is the address of a **DDCAPS** that will be filled with information describing the device driver's capabilities. Because the application doesn't need information about emulation capabilities, the second parameter is set to NULL.

After retrieving the driver capabilities, the example checks the **dwCaps** member for the presence of the DDCAPS_OVERLAY flag using a logical **AND** operation. If the flag isn't present, the example returns FALSE to indicate failure. Otherwise, the example returns TRUE to indicate that the device driver supports overlay surfaces.

In your code, this might be a good time for you to check the **dwMaxVisibleOverlays** and **dwCurrentVisibleOverlays** members in the DDCAPS structure to ensure that no other overlay surfaces are in use by other applications.

Step 3: Creating an Overlay Surface

Now that you know that the driver supports overlay surfaces, you can try to create one. Because there is no standard dictating how devices must support overlay surfaces, you can't count on being able to create overlays of any particular size or pixel format. Additionally, you can't expect to succeed in creating an overlay surface on the first try. Therefore, be prepared to attempt creation multiple times starting with the most desirable characteristics, falling back on less desirable (but possibly less hardware intensive) configurations until one works.

(You can call the **IDirectDraw2::GetFourCCCodes** method to retrieve a list of FOURCC codes that describe non-RGB pixel formats that the driver will likely support for overlay surfaces. However, in you want to try using RGB overlay surfaces, it is recommended that you attempt to creating surfaces in various common RGB formats, falling back on another format if you fail.)

The Mosquito sample follows a "best case to worst case" philosophy when creating an overlay surface. Mosquito first tries to create a triple-buffered page flipping complex overlay surface. If the creation attempt fails, the sample tries the configuration with other common pixel formats. The following code fragment shows how this can be done:

```
    ZeroMemory(&ddsdOverlay, sizeof(ddsdOverlay));
    ddsdOverlay.dwSize = sizeof(ddsdOverlay);

    ddsdOverlay.dwFlags= DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH |
                         DDSD_BACKBUFFERCOUNT| DDSD_PIXELFORMAT;
    ddsdOverlay.ddsCaps.dwCaps = DDSCAPS_OVERLAY | DDSCAPS_FLIP |
                                 DDSCAPS_COMPLEX | DDSCAPS_VIDEOMEMORY;
    ddsdOverlay.dwWidth  =320;
    ddsdOverlay.dwHeight =240;
    ddsdOverlay.dwBackBufferCount=2;

    // Try to create an overlay surface using one of the pixel formats in
our
    // global list.
    i=0;
    do{
        ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
       // Try to create the overlay surface
       ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
    } while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );
```

The preceding example sets the flags and values within a **DDSURFACEDESC** structure to reflect a triple-buffered page flipping complex overlay surface. Then, the sample performs a loop during which it attempts to create the requested surface in a variety of common pixel formats, in order of most desirable to least desirable pixel formats. If the attempt succeeds, the loop ends. If all the attempts fail, it's likely that the display hardware doesn't have enough memory to support a triple-buffered scheme or that it doesn't support flipping overlay surfaces. In this case, the sample falls back on a less desirable configuration using a single non-flipping overlay surface, as shown in the following example:

```
    // If we failed to create a triple buffered complex overlay surface, try
    // again with a single non-flippable buffer.
    if(FAILED(ddrval))
    {
        ddsdOverlay.dwBackBufferCount=0;
        ddsdOverlay.ddsCaps.dwCaps=DDSCAPS_OVERLAY | DDSCAPS_VIDEOMEMORY;
        ddsdOverlay.dwFlags= DDSD_CAPS|DDSD_HEIGHT|DDSD_WIDTH|
DDSD_PIXELFORMAT;
```

```
        // Try to create the overlay surface
        ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay, NULL);
        i=0;
        do{
            ddsdOverlay.ddpfPixelFormat=g_ddpfOverlayFormats[i];
            ddrval = g_lpdd->CreateSurface(&ddsdOverlay, &g_lpddsOverlay,
NULL);
        } while( FAILED(ddrval) && (++i < NUM_OVERLAY_FORMATS) );

        // We couldn't create an overlay surface.  Exit, returning failure.
        if (FAILED(ddrval))
            return FALSE;
    }
```

The code above resets the flags and values in the **DDSURFACEDESC** structure to reflect a single non-flipping overlay surface. Again, the example loops through pixel formats attempting to create the surfaces, stopping the loop if an attempt succeeded. If the attempts still didn't work, the sample returns FALSE to indicate failure.

After you've successfully created your overlay surface or surfaces, you can load bitmaps onto them in preparation for display.

Step 4: Displaying the Overlay Surface

After creating your overlay surface, you can display it. Often, display hardware imposes alignment restrictions on the position and pixel width of the rectangles you use to display the overlay. Additionally, you will often need to account for a minimum required stretch factor by adjusting the width of the destination rectangle in order to successfully display the overlay surface. The Mosquito sample performs the following tasks to prepare and display the overlay surface:

- Step 4.1: Determining the Minimum Display Requirements
- Step 4.2: Setting Up the Source and Destination Rectangles
- Step 4.3: Displaying the Overlay Surface

Step 4.1: Determining the Minimum Display Requirements

Most display hardware imposes restrictions on displaying overlay surfaces. You must carefully meet these restrictions in order to successfully display an overlay surface. You can retrieve information about these restrictions by calling the **IDirectDraw2::GetCaps** method. The **DDCAPS** structure that the method fills contains information about overlay capabilities and their usage restrictions. Hardware restrictions vary, so always look at the flags included in the **dwFlags** member to determine which restrictions apply to you.

The Mosquito sample starts by retrieving the hardware capabilities, then takes action based upon the minimum stretch factor, as shown in the following code fragment:

```
    // Get driver capabilities
    ddrval = g_lpdd->GetCaps(&capsDrv, NULL);
    if (FAILED(ddrval))
        return FALSE;

    // Check the minimum stretch and set the local variable accordingly.
    if(capsDrv.dwCaps & DDCAPS_OVERLAYSTRETCH)
        uStretchFactor1000 = (capsDrv.dwMinOverlayStretch>1000) ?
capsDrv.dwMinOverlayStretch : 1000;
    else
        uStretchFactor1000 = 1000;
```

The code above calls **IDirectDraw2::GetCaps** to retrieve only the hardware capabilities. For this call, the first parameter is a pointer the **DDCAPS** structure that will be filled with the capability information for the device driver, and the second parameter is NULL to indicate that emulation information is not to be retrieved.

The example retains the minimum stretch factor in a temporary variable for use later. (Keep in mind that stretch factors are reported multiplied by 1000, so 1300 really means 1.3.) If the driver reports a value greater than 1000, it means that the driver requires that all destination rectangles must be stretched along the X-axis by a ratio of the reported value. For example, if the driver reports a stretch factor 1.3 and the source rectangle is 320 pixels wide, the destination rectangle must be at least 416 pixels wide. If the driver reports a stretch factor less than 1000, it means that the driver can display overlays smaller than the source rectangle, but can also stretch the overlay if desired.

Next, the sample examines values describing the driver's size alignment restrictions, as shown in the following example:

```
    // Grab any alignment restrictions and set the local variables
acordingly.
    uSrcSizeAlign = (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
capsDrv.dwAlignSizeSrc:0;
    uDestSizeAlign= (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC)?
capsDrv.dwAlignSizeDest:0;
```

The sample uses more temporary variables to hold the reported size alignment restrictions taken from the **dwAlignSizeSrc** and **dwAlignSizeDest** members. These values provide information about pixel width alignment restrictions and are needed when setting the dimensions of the source and destination rectangles to reflect these restrictions later. Source and destination rectangles must have a pixel width that is a multiple of the values in these members.

Last, the sample examines the value that describes the destination rectangle boundary alignment:

```
    // Set the "destination position alignment" global so we won't have to
    // keep calling GetCaps() every time we move the overlay surface.
    if (capsDrv.dwCaps & DDCAPS_ALIGNBOUNDARYDEST)
```

```
        g_dwOverlayXPositionAlignment = capsDrv.dwAlignBoundaryDest;
    else
        g_dwOverlayXPositionAlignment = 0;
```

The preceding code uses a global variable to hold the value for the destination rectangle's boundary alignment, as taken from the **dwAlignBoundaryDest** member. This value will be used when the program repositions the overlay later. (For details, see Step 5: Updating the Overlay Display Position) You must set the x-coordinate of the destination rectangle's top left corner to be aligned with this value, in pixels. That is, if the value specified is 4, you can only specify destination rectangles whose top-left corner has an x-coordinate at pixels 0, 4, 8, 12, and so on. The Mosquito application initially displays the overlay at 0,0, so alignment compliance is assumed and the sample doesn't need to retrieve the restriction information until after displaying the overlay the first time. Your implementation might vary, so you will probably need to check this information and adjust the destination rectangle before displaying the overlay.

Step 4.2: Setting Up the Source and Destination Rectangles

After retrieving the driver's overlay restrictions you should set the values for your source and destination rectangles accordingly, assuring that you will be able to successfully display the overlay. The following sample from the Mosquito sample application starts by setting the characteristics of the source rectangle:

```
    // Set initial values in the source RECT.
    rs.left=0; rs.top=0;
    rs.right = 320;
    rs.bottom = 240;

    // Apply size alignment restrictions, if necessary.
    if (capsDrv.dwCaps & DDCAPS_ALIGNSIZESRC && uSrcSizeAlign)
        rs.right -= rs.right % uSrcSizeAlign;
```

The preceding code sets initial values for the surface to include the dimensions of the entire surface. If the device driver requires size alignment for the source rectangle, the example adjusts the source rectangle to conform. The example adjusts the width of the source rectangle to be narrower than the original size because the width cannot be expanded without completely recreating the surface. However, your code could just as easily start with a smaller rectangle and widen the rectangle to meet driver restrictions.

After the dimensions of the source rectangle are set and conform with hardware restrictions, you need to set and adjust the dimensions of the destination rectangle. This process requires a little more work because the rectangle might need to be stretched first, then adjusted to meet size alignment restrictions. The following code performs the task of accounting for the minimum stretch factor:

```
    // Set up the destination RECT, starting with the source RECT values.
    // We use the source RECT dimensions instead of the surface dimensions
in
    // case they differ.
    rd.left=0; rd.top=0;
    rd.right  = (rs.right*uStretchFactor1000+999)/1000;    // (Adding 999
avoids integer truncation problems.)

    // (This isn't required by DDraw, but we'll stretch the
    //  height, too, to maintain aspect ratio).
    rd.bottom = rs.bottom*uStretchFactor1000/1000;
```

The preceding code sets the top left corner of the destination rectangle to the top left corner of the screen, then sets the width to account for the minimum stretch factor. While adjusting for the stretch factor, note that the example adds 999 to the product of the width and stretch factor. This is done to prevent integer truncation that could result in a rectangle that isn't as wide as the minimum stretch factor requires. For more information, see Minimum and Maximum Stretch Factors. Also, after the example stretches the width, it stretches the height. Stretching the height isn't required, but was done to preserve the bitmap's aspect ratio and avoid a distorted appearance.

After stretching the destination rectangle, the example continues by adjusting it to conform to size alignment restrictions as follows:

```
    // Adjust the destination RECT's width to comply with any imposed
    // alignment restrictions.
    if (capsDrv.dwCaps & DDCAPS_ALIGNSIZEDEST && uDestSizeAlign)
        rd.right =
(int)((rd.right+uDestSizeAlign-1)/uDestSizeAlign)*uDestSizeAlign;
```

The example checks the capabilities flags to see if the driver imposes destination size alignment

restrictions. If so, the destination rectangle's width is increased by enough pixels to meet alignment restrictions. Note that the rectangle is adjusted by expanding the width, not by decreasing it. This is done because decreasing the width could cause the destination rectangle to be smaller than is required by the minimum stretch factor, consequently causing attempts to display the overlay surface to fail.

Step 4.3: Displaying the Overlay Surface

After you've set up the source and destination rectangles, you can display the overlay for the first time. If you've prepared correctly, this will be simple. The Mosquito sample uses the following code to initially display the overlay:

```
    // Set the flags we'll send to UpdateOverlay
    dwUpdateFlags = DDOVER_SHOW | DDOVER_DDFX;

    // Does the overlay hardware support source color keying?
    // If so, we can hide the black background around the image.
    // This probably won't work with YUV formats
    if (capsDrv.dwCKeyCaps & DDCKEYCAPS_SRCOVERLAY)
        dwUpdateFlags |= DDOVER_KEYSRCOVERRIDE;

    // Create an overlay FX structure so we can specify a source color key.
    // This information is ignored if the DDOVER_SRCKEYOVERRIDE flag isn't
set.
    ZeroMemory(&ovfx, sizeof(ovfx));
    ovfx.dwSize = sizeof(ovfx);

    ovfx.dckSrcColorkey.dwColorSpaceLowValue=0; // Specify black as the
color key
    ovfx.dckSrcColorkey.dwColorSpaceHighValue=0;

    // Call UpdateOverlay() to displays the overlay on the screen.
    ddrval = g_lpddsOverlay->UpdateOverlay(&rs, g_lpddsPrimary, &rd,
dwUpdateFlags, &ovfx);
    if(FAILED(ddrval))
        return FALSE;
```

The preceding example starts by setting the DDOVER_SHOW and DDOVER_DDFX flags in the *dwUpdateFlags* temporary variable, indicating that the overlay is to be displayed for the first time, and that the hardware should use the effects information included in an associated **DDOVERLAYFX** structure to do so. Next, the example checks a previously existing **DDCAPS** structure to determine if overlay source color keying is supported. If it is, the DDOVER_KEYSRCOVERRIDE is included in the *dwUpdateFlags* variable to take advantage of source color keying and the example sets color key values accordingly.

After preparation is complete, the example calls the **IDirectDrawSurface3::UpdateOverlay** method to display the overlay. For the call, the first and third parameters are the addresses of the adjusted source and destination rectangles. The second parameter is the address of the primary surface over which the overlay will be displayed. The fourth parameter consists of the flags placed in the previously prepared *dwUpdateFlags* variable, and the fifth parameter is the address of **DDOVERLAYFX** structure whose members were set to match those flags.

If the hardware only supports one overlay surface and that surface is in use, the **UpdateOverlay** method fails, returning DDERR_OUTOFCAPS. Additionally, if **UpdateOverlay** fails, you might try increasing the width of the destination rectangle to accommodate for the possibility that the hardware incorrectly reported a minimum stretch factor that was too small. However, this rarely occurs and Mosquito simply fails if **UpdateOverlay** doesn't succeed.

Step 5: Updating the Overlay Display Position

After displaying the overlay surface, you might not need to do anything else. However, some software might need to reposition the overlay surface. The Mosquito sample uses the **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay, as shown in the following example.

```
    // Set X- and Y-coordinates
    .
    .
    .
    // We need to check for any alignment restrictions on the X position
    // and align it if necessary.
    if (g_dwOverlayXPositionAlignment)
        dwXAligned = g_nOverlayXPos - g_nOverlayXPos %
g_dwOverlayXPositionAlignment;
    else
        dwXAligned = g_nOverlayXPos;

    // Set the overlay to its new position.
    ddrval = g_lpddsOverlay->SetOverlayPosition(dwXAligned, g_nOverlayYPos);
    if (ddrval == DDERR_SURFACELOST)
    {
        if (!RestoreAllSurfaces())
            return;
    }
```

The preceding example starts by aligning the rectangle to meet any destination rectangle boundary alignment restrictions that might exist. The global variable that it checks, *g_dwOverlayXPositionAlignment*, was set earlier to equal the value reported in the **dwAlignBoundaryDest** member of the **DDCAPS** structure when the application previously called the **IDirectDraw2::GetCaps** method. (For details, see Step 4.1: Determining the Minimum Display Requirements). If destination alignment restrictions exist, the example adjusts the new x-coordinate to be pixel-aligned accordingly. Failing to meet this requirement will cause the overlay surface not to be displayed.

After making any requisite adjustments to the new x-coordinate, the example calls **IDirectDrawSurface3::SetOverlayPosition** method to reposition the overlay. For the call, the first parameter is the aligned x-coordinate, and the second parameter is the new y-coordinate. These values represent the new location of the overlay's top-left corner. Width and height information are not accepted, nor are they needed because DirectDraw already knows the dimensions of the surface from the **IDirectDrawSurface3::UpdateOverlay** method made to initially display the overlay. If the call fails because one or more surfaces were lost, the example calls an application-defined function to restore them and reload their bitmaps.

Note: Take care not to use coordinates too close to the bottom or right edge of the target surface. The **IDirectDraw2::SetOverlayPosition** method does not perform clipping for you; using coordinates that would potentially make the overlay run off the edge of the target surface will cause the method to fail, returning DDERR_INVALIDPOSITION.

Step 6: Hiding the Overlay Surface

When you do not need the overlay surface anymore, or if you simply want to remove it from view, you can hide the surface by calling the **IDirectDrawSurface3::UpdateOverlay** method with appropriate flags. Mosquito hides the overlay in preparation for closing the application using the following code:

```
void DestroyOverlay()
{
    if (g_lpddsOverlay){
        // Use UpdateOverlay() with the DDOVER_HIDE flag to remove an
overlay
        // from the display.
        g_lpddsOverlay->UpdateOverlay(NULL, g_lpddsPrimary, NULL,
DDOVER_HIDE, NULL);
        g_lpddsOverlay->Release();
        g_lpddsOverlay=NULL;
    }
}
```

When the preceding example calls **IDirectDrawSurface3::UpdateOverlay**, it specifies NULL for the source and destination rectangles, because they are irrelevant when hiding the overlay. Similarly, the example uses NULL in the fourth parameter because overlay effects aren't being used. The second parameter is a pointer to the target surface. Lastly, the example uses the DDOVER_HIDE flag in the fourth parameter to indicate that the overlay will be removed from view.

After the example hides the overlay, the example releases its **IDirectDrawSurface3** interface and invalidates its global variable by setting it to NULL. For the purposes of the Mosquito sample application, the overlay surface is no longer needed. If you still need the overlay surface for later, you could simply hide the overlay without releasing it, then redisplay it whenever you require.

Other DirectDraw Samples

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

- Stretch

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

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

  Demonstrates palette animation.
- Dxview

  Demonstrates how to retrieve the capabilities of the display hardware.

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

DirectDraw Reference

This section contains reference information for the API elements that DirectDraw provides. Reference material is divided into the following categories:

- Interfaces
- Functions
- Callback Functions
- Structures
- Return Values
- Pixel Format Masks
- Four Character Codes (FOURCC)

Interfaces

This section contains reference information about the interfaces used with the DirectDraw component. The following interfaces are covered:

- **IDDVideoPortContainer**
- **IDirectDraw2**
- **IDirectDrawClipper**
- **IDirectDrawColorControl**
- **IDirectDrawPalette**
- **IDirectDrawSurface3**
- **IDirectDrawVideoPort**

IDDVideoPortContainer

Applications use the methods of the **IDDVideoPortContainer** interface to create and manipulate DirectDrawVideoPort objects.

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

Creating objects	**CreateVideoPort**
Video ports	**EnumVideoPorts** **QueryVideoPortStatus**
Connections	**GetVideoPortConnectInfo**

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

AddRef
QueryInterface
Release

You can use the LPDDVIDEOPORTCONTAINER data type to declare a variable that contains a pointer to an **IDDVideoPortContainer** interface. The Dvp.h header file declares the LPDDVIDEOPORTCONTAINER with the following code:

```
typedef struct IDDVideoPortContainer    FAR *LPDDVIDEOPORTCONTAINER;
```

IDDVideoPortContainer::CreateVideoPort

The **IDDVideoPortContainer::CreateVideoPort** method creates a DirectDrawVideoPort object.

```
HRESULT CreateVideoPort(
  DWORD dwFlags,
  LPDDVIDEOPORTDESC lpDDVideoPortDesc,
  LPDIRECTDRAWVIDEOPORT FAR *lplpDDVideoPort,
  IUnknown FAR *pUnkOuter
);
```

Parameters

dwFlags
Reserved for future use. This parameter must be zero.

lpDDVideoPortDesc
Address of a **DDVIDEOPORTDESC** structure that describes the VideoPort object to be created.

lplpDDVideoPort
Address of a variable that will be filled with a pointer to the new DirectDrawVideoPort object's **IDirectDrawVideoPort** interface if the call succeeds.

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFCAPS
DDERR_OUTOFMEMORY

IDDVideoPortContainer::EnumVideoPorts

The **IDDVideoPortContainer::EnumVideoPorts** method enumerates all of the video ports that the hardware exposes that are compatible with a provided video port description.

```
HRESULT EnumVideoPorts(
  DWORD dwFlags,
  LPDDVIDEOPORTCAPS lpDDVideoPortCaps,
  LPVOID lpContext,
  LPENUMVIDEOCALLBACK lpEnumVideoCallback
);
```

Parameters

dwFlags
Reserved for future use. This parameter must be zero.

lpDDVideoPortCaps
Pointer to a **DDVIDEOPORTCAPS** structure that will be checked against the available video ports. If this parameter is NULL, all video ports will be enumerated.

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

lpEnumVideoCallback
Address of the **EnumVideoCallback** function that will be called each time a match is found.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

IDDVideoPortContainer::GetVideoPortConnectInfo

The **IDDVideoPortContainer::GetVideoPortConnectInfo** method retrieves the connection information supported by all video ports.

```
HRESULT GetVideoPortConnectInfo(
  DWORD dwPortId,
  LPDWORD lpNumEntries,
  LPDDVIDEOPORTCONNECT lpConnectInfo
);
```

Parameters

dwPortId

Video port ID of the video port for which the connection information will be retrieved.

lpNumEntries

Address of a variable containing the number of entries that the array at *lpConnectInfo* can hold. If this number is less than the total number of connections, the method fills the array with as many entries as will fit, sets the value at *lpNumEntries* to indicate the total number of connections, and returns DDERR_MOREDATA.

lpConnectInfo

Address of an array of **DDVIDEOPORTCONNECT** structures that will be filled with the connection options supported by the specified video port. If this parameter is NULL, the method sets *lpNumEntries* to indicate the total number of connections that the video port supports, then returns DD_OK.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_MOREDATA

IDDVideoPortContainer::QueryVideoPortStatus

The **IDDVideoPortContainer::QueryVideoPortStatus** method is not currently implemented.

```
HRESULT QueryVideoPortStatus(
  DWORD dwPortId,
  LPDDVIDEOPORTSTATUS lpVPStatus
);
```

Parameters

dwPortId

Video port ID of the video port for which the status information will be retrieved.

lpVPStatus

Address of a **DDVIDEOPORTSTATUS** structure that will be filled with information about the status of the specified video port.

Return Values

This method returns DDERR_UNSUPPORTED.

IDirectDraw2

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

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

Allocating memory	**Compact** **Initialize**
Creating objects	**CreateClipper** **CreatePalette** **CreateSurface**
Device capabilities	**GetCaps**
Display modes	**EnumDisplayModes** **GetDisplayMode** **GetMonitorFrequency** **RestoreDisplayMode** **SetDisplayMode** **WaitForVerticalBlank**
Display status	**GetScanLine** **GetVerticalBlankStatus**
Miscellaneous	**GetAvailableVidMem** **GetFourCCCodes**
Setting behavior	**SetCooperativeLevel**
Surfaces	**DuplicateSurface** **EnumSurfaces** **FlipToGDISurface** **GetGDISurface**

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

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAW or LPDIRECTDRAW data types to declare a variable that contains a pointer to an **IDirectDraw** or **IDirectDraw2** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDraw     FAR *LPDIRECTDRAW;
typedef struct IDirectDraw2    FAR *LPDIRECTDRAW2;
```

IDirectDraw2::Compact

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

HRESULT Compact();

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_SURFACEBUSY

Remarks

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

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

IDirectDraw2::CreateClipper

The **IDirectDraw2::CreateClipper** method creates a DirectDrawClipper object.

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

Parameters

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

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOOPERATIVELEVELSET
DDERR_OUTOFMEMORY

Remarks

The DirectDrawClipper object can be attached to a DirectDrawSurface and used during **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltBatch**, and **IDirectDrawSurface3::UpdateOverlay** operations.

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

See Also

IDirectDrawSurface3::GetClipper, **IDirectDrawSurface3::SetClipper**

IDirectDraw2::CreatePalette

The **IDirectDraw2::CreatePalette** method creates a DirectDrawPalette object for this DirectDraw object.

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

Parameters

dwFlags

One or more of the following flags:

DDPCAPS_1BIT	Indicates that the index is 1 bit. There are two entries in the color table.
DDPCAPS_2BIT	Indicates that the index is 2 bits. There are four entries in the color table.
DDPCAPS_4BIT	Indicates that the index is 4 bits. There are 16 entries in the color table.
DDPCAPS_8BITENTRIES	Indicates that the index refers to an 8-bit color index. This flag is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT flag, and when the target surface is in 8 bpp. Each color entry is 1 byte long and is an index to a destination surface's 8-bpp palette.
DDPCAPS_8BIT	Indicates that the index is 8 bits. There are 256 entries in the color table.
DDPCAPS_ALLOW256	Indicates that this palette can have all 256 entries defined.

lpColorTable

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

lplpDDPalette

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

pUnkOuter

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NOCOOPERATIVELEVELSET

DDERR_OUTOFMEMORY

DDERR_UNSUPPORTED

IDirectDraw2::CreateSurface

The **IDirectDraw2::CreateSurface** method creates a DirectDrawSurface object for this DirectDraw object.

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

Parameters

lpDDSurfaceDesc
Address of the **DDSURFACEDESC** structure that describes the requested surface. You should set any unused members of **DDSURFACEDESC** to zero before calling this method. A **DDSCAPS** structure is a member of **DDSURFACEDESC**.

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

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_NOOVERLAYHW
DDERR_NOZBUFFERHW
DDERR_OUTOFMEMORY
DDERR_OUTOFVIDEOMEMORY
DDERR_PRIMARYSURFACEALREADYEXISTS
DDERR_UNSUPPORTEDMODE

IDirectDraw2::DuplicateSurface

The **IDirectDraw2::DuplicateSurface** method duplicates a DirectDrawSurface object.

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

Parameters

lpDDSurface
Address of the DirectDrawSurface structure to be duplicated.

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTDUPLICATE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY
DDERR_SURFACELOST

Remarks

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, 3-D surface, or implicitly created surface cannot be duplicated.

IDirectDraw2::EnumDisplayModes

The **IDirectDraw2::EnumDisplayModes** method enumerates all of the display modes the hardware exposes through the DirectDraw object that are compatible with a provided surface description.

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

Parameters

dwFlags

DDEDM_REFRESHRATES	Enumerates modes with different refresh rates. **IDirectDraw2::EnumDisplayModes** guarantees that a particular mode will be enumerated only once. This flag specifies whether the refresh rate is taken into account when determining if a mode is unique.
DDEDM_STANDARDVGAMODES	Enumerates Mode 13 in addition to the 320x200x8 Mode X mode.

lpDDSurfaceDesc

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

lpContext

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

lpEnumModesCallback

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

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

See Also

IDirectDraw2::GetDisplayMode, **IDirectDraw2::SetDisplayMode**, **IDirectDraw2::RestoreDisplayMode**

IDirectDraw2::EnumSurfaces

The **IDirectDraw2::EnumSurfaces** method enumerates all of the existing or possible surfaces that meet the search criterion specified.

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

Parameters

dwFlags

One of the following flags:

DDENUMSURFACES_ALL

Enumerates all of the surfaces that meet the search criterion.

DDENUMSURFACES_CANBECREATED

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

DDENUMSURFACES_DOESEXIST

Enumerates the already existing surfaces that meet the search criterion.

DDENUMSURFACES_MATCH

Searches for any surface that matches the surface description.

DDENUMSURFACES_NOMATCH

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

lpDDSD

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

lpContext

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

lpEnumSurfacesCallback

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

If the DDENUMSURFACES_CANBECREATED flag is set, this method attempts to temporarily create a surface that meets the criterion. Note that as a surface is enumerated, its reference count is

increased—if you are not going to use the surface, use **IDirectDraw::Release** to release the surface after each enumeration.

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

IDirectDraw2::FlipToGDISurface

The **IDirectDraw2::FlipToGDISurface** method makes the surface that GDI writes to the primary surface.

```
HRESULT FlipToGDISurface();
```

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

Remarks

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

See Also

IDirectDraw2::GetGDISurface

IDirectDraw2::GetAvailableVidMem

The **IDirectDraw2::GetAvailableVidMem** method retrieves the total amount of display memory available and the amount of display memory currently free for a given type of surface.

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

Parameters

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

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

lpdwFree
Address of a variable that will be filled with the amount of display memory currently free that can be allocated for a surface that matches the capabilities specified by the structure at *lpDDSCaps*.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDCAPS
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NODIRECTDRAWHW

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

Remarks

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

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

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

This method provides only a snapshot of the current display-memory state. The amount of free display memory is subject to change as surfaces are created and released. Therefore, you should use the free memory value only as an approximation. In addition, a particular display adapter card may make no distinction between two different memory types. For example, the adapter might use the same portion of display memory to store z-buffers and textures. So, allocating one type of surface (for example, a z-buffer) can affect the amount of display memory available for another type of surface (for example, textures). Therefore, it is best to first allocate an application's fixed resources (such as front and back buffers , and z-buffers) before determining how much memory is available for dynamic use (such as texture mapping).

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

IDirectDraw2::GetCaps

The **IDirectDraw2::GetCaps** method fills in the capabilities of the device driver for the hardware and the hardware-emulation layer (HEL).

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

Parameters

lpDDDriverCaps

Address of a **DDCAPS** structure that will be filled with the capabilities of the hardware, as reported by the device driver. Set this parameter to NULL if device driver capabilities are not to be retrieved.

lpDDHELCaps

Address of a **DDCAPS** structure that will be filled with the capabilities of the HEL. Set this parameter to NULL if HEL capabilities are not to be retrieved.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

You can only set one of the two parameters to NULL to exclude it. If you set both to NULL the method will fail, returning DDERR_INVALIDPARAMS.

IDirectDraw2::GetDisplayMode

The **IDirectDraw2::GetDisplayMode** method retrieves the current display mode.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTEDMODE

Remarks

An application should not save the information returned by this method to restore the display mode on clean-up. The application should use the **IDirectDraw2::RestoreDisplayMode** method to restore the mode on clean-up, thereby avoiding mode-setting conflicts that could arise in a multiprocess environment.

See Also

IDirectDraw2::SetDisplayMode, **IDirectDraw2::RestoreDisplayMode**, **IDirectDraw2::EnumDisplayModes**

IDirectDraw2::GetFourCCCodes

The **IDirectDraw2::GetFourCCCodes** method retrieves the FOURCC codes supported by the DirectDraw object. This method can also retrieve the number of codes supported.

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

Parameters

lpNumCodes

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

lpCodes

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

IDirectDraw2::GetGDISurface

The **IDirectDraw2::GetGDISurface** method retrieves the DirectDrawSurface object that currently represents the surface memory that GDI is treating as the primary surface.

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

Parameters

lplpGDIDDSSurface
Address of a DirectDrawSurface pointer to the DirectDrawSurface object that currently controls GDIs primary surface memory.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND

See Also

IDirectDraw2::FlipToGDISurface

IDirectDraw2::GetMonitorFrequency

The **IDirectDraw2::GetMonitorFrequency** method retrieves the frequency of the monitor being driven by the DirectDraw object.

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

Parameters

lpdwFrequency
Address of the variable that will be filled with the monitor frequency, reported in Hz.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

IDirectDraw2::GetScanLine

The **IDirectDraw2::GetScanLine** method retrieves the scan line that is currently being drawn on the monitor.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS

Remarks

Scan lines are reported as zero-based integers. The returned scan line value is between 0 and *n*, where scan line 0 is the first visible scan line on the screen and *n* is the last visible scan line, plus any scan lines that occur during vblank. So, in a case where an application is running at 640480, and there are 12 scan lines during vblank, the values returned by this method will range from 0 to 491.

See Also

IDirectDraw2::GetVerticalBlankStatus, **IDirectDraw2::WaitForVerticalBlank**

IDirectDraw2::GetVerticalBlankStatus

The **IDirectDraw2::GetVerticalBlankStatus** method retrieves the status of the vertical blank.

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

Parameters

lpbIsInVB

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

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

See Also

IDirectDraw2::GetScanLine, **IDirectDraw2::WaitForVerticalBlank**

IDirectDraw2::Initialize

The **IDirectDraw2::Initialize** method initializes the DirectDraw object that was created by using the **CoCreateInstance** COM function.

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

Parameters

lpGUID

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_ALREADYINITIALIZED

DDERR_DIRECTDRAWALREADYCREATED

DDERR_GENERIC

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NODIRECTDRAWHW

DDERR_NODIRECTDRAWSUPPORT

DDERR_OUTOFMEMORY

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

Remarks

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

IDirectDraw2::RestoreDisplayMode

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

HRESULT RestoreDisplayMode();

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES
DDERR_NOEXCLUSIVEMODE

See Also

IDirectDraw2::SetDisplayMode, **IDirectDraw2::EnumDisplayModes**, **IDirectDraw2::SetCooperativeLevel**

IDirectDraw2::SetCooperativeLevel

The **IDirectDraw2::SetCooperativeLevel** method determines the top-level behavior of the application.

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

Parameters

hWnd

Window handle used for the application. This parameter can be NULL when the DDSCL_NORMAL flag is specified in the *dwFlags* parameter.

dwFlags

One or more of the following flags:

DDSCL_ALLOWMODEX

Allows the use of Mode X display modes. This flag must be used with the DDSCL_EXCLUSIVE and DDSCL_FULLSCREEN flags.

DDSCL_ALLOWREBOOT

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

DDSCL_EXCLUSIVE

Requests the exclusive level. This flag must be used with the DDSCL_FULLSCREEN flag.

DDSCL_FULLSCREEN

Indicates that the exclusive-mode owner will be responsible for the entire primary surface. GDI can be ignored. This flag must be used with the DDSCL_EXCLUSIVE flag.

DDSCL_NORMAL

Indicates that the application will function as a regular Windows application. This flag cannot be used with the DDSCL_ALLOWMODEX, DDSCL_EXCLUSIVE, or DDSCL_FULLSCREEN flags.

DDSCL_NOWINDOWCHANGES

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCLUSIVEMODEALREADYSET

DDERR_HWNDALREADYSET

DDERR_HWNDSUBCLASSED

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

An application must set either the DDSCL_EXCLUSIVE or DDSCL_NORMAL flag.

The DDSCL_EXCLUSIVE flag must be set to call functions that can have drastic performance consequences for other applications. For more information, see Cooperative Levels.

Interaction between this method and the **IDirectDraw2::SetDisplayMode** method differs from their **IDirectDraw** counterparts. For more information, see Cooperative Levels and Display Modes with IDirectDraw2.

See Also

IDirectDraw2::SetDisplayMode, **IDirectDraw2::Compact**, **IDirectDraw2::EnumDisplayModes**, Mode X and Mode 13 Display Modes.

IDirectDraw2::SetDisplayMode

The **IDirectDraw2::SetDisplayMode** method sets the mode of the display-device hardware.

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

Parameters

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

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

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

dwFlags
Flags describing additional options. Currently, the only valid flag is DDSDM_STANDARDVGAMODE, which causes the method to set Mode 13 instead of Mode X 320x200x8 mode. If you are setting another resolution, bit depth, or a Mode X mode, do not use this flag and set the parameter to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDMODE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_LOCKEDSURFACES
DDERR_NOEXCLUSIVEMODE
DDERR_SURFACEBUSY
DDERR_UNSUPPORTED
DDERR_UNSUPPORTEDMODE
DDERR_WASSTILLDRAWING

Remarks

If another application changes the display mode, the primary surface will be lost and will return DDERR_SURFACELOST until it is recreated to match the new display mode.

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

See Also

IDirectDraw2::RestoreDisplayMode, **IDirectDraw2::GetDisplayMode**, **IDirectDraw2::EnumDisplayModes**, **IDirectDraw2::SetCooperativeLevel**, Cooperative Levels and

Display Modes with IDirectDraw2, Setting Display Modes

IDirectDraw2::WaitForVerticalBlank

The **IDirectDraw2::WaitForVerticalBlank** method helps the application synchronize itself with the vertical-blank interval.

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

Parameters

dwFlags

Determines how long to wait for the vertical blank.

DDWAITVB_BLOCKBEGIN

Returns when the vertical-blank interval begins.

DDWAITVB_BLOCKBEGINEVENT

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

DDWAITVB_BLOCKEND

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

hEvent

Handle of the event to be triggered when the vertical blank begins. This parameter is not currently used.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

See Also

IDirectDraw2::GetVerticalBlankStatus, **IDirectDraw2::GetScanLine**

IDirectDrawClipper

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

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

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

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

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAWCLIPPER data type to declare a variable that contains a pointer to an **IDirectDrawClipper** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawClipper    FAR *LPDIRECTDRAWCLIPPER;
```

IDirectDrawClipper::GetClipList

The **IDirectDrawClipper::GetClipList** method retrieves a copy of the clip list associated with a DirectDrawClipper object. A subset of the clip list can be selected by passing a rectangle that clips the clip list.

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

Parameters

lpRect

Address of a rectangle that will be used to clip the clip list. This parameter can be NULL to retrieve the entire clip list.

lpClipList

Address of an **RGNDATA** structure that will contain the resulting copy of the clip list. If this parameter is NULL, the method fills the variable at *lpdwSize* to the number of bytes necessary to hold the entire clip list.

lpdwSize

Size of the resulting clip list. When retrieving the clip list, this parameter is the size of the buffer at *lpClipList*. When *lpClipList* is NULL, the variable at *lpdwSize* receives the required size of the buffer, in bytes.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPLIST
DDERR_REGIONTOOSMALL

Remarks

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

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

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

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

```
} RGNDATAHEADER;
```

For more information about these structures, see the documentation in the Platform SDK.

See Also

IDirectDrawClipper::SetClipList

IDirectDrawClipper::GetHWnd

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

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

Parameters

lphWnd

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

See Also

IDirectDrawClipper::SetHWnd

IDirectDrawClipper::Initialize

The **IDirectDrawClipper::Initialize** method initializes a DirectDrawClipper object that was created by using the **CoCreateInstance** COM function.

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

Parameters

lpDD

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

dwFlags

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_ALREADYINITIALIZED

DDERR_INVALIDPARAMS

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

Remarks

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

See Also

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

IDirectDrawClipper::IsClipListChanged

The **IDirectDrawClipper::IsClipListChanged** method monitors the status of the clip list if a window handle is associated with a DirectDrawClipper object.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

IDirectDrawClipper::SetClipList

The **IDirectDrawClipper::SetClipList** method sets or deletes the clip list used by the **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltBatch**, and **IDirectDrawSurface3::UpdateOverlay** methods on surfaces to which the parent DirectDrawClipper object is attached.

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

Parameters

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CLIPPERISUSINGHWND
DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

The clip list cannot be set if a window handle is already associated with the DirectDrawClipper object. Note that the **IDirectDrawSurface3::BltFast** method cannot clip.

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

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

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

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

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

See Also

IDirectDrawClipper::GetClipList, **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltFast**, **IDirectDrawSurface3::BltBatch**, **IDirectDrawSurface3::UpdateOverlay**

IDirectDrawClipper::SetHWnd

The **IDirectDrawClipper::SetHWnd** method sets the window handle that will obtain the clipping information.

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

Parameters

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

hWnd
Window handle that obtains the clipping information.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDCLIPLIST
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

See Also

IDirectDrawClipper::GetHWnd

IDirectDrawColorControl

The **IDirectDrawColorControl** interface allows you to get and set color controls:

Color controls	**GetColorControls**
	SetColorControls

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

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAWCOLORCONTROL data type to declare a variable that contains a pointer to an **IDirectDrawColorControl** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawColorControl    FAR *LPDIRECTDRAWCOLORCONTROL;
```

IDirectDrawColorControl::GetColorControls

The **IDirectDrawColorControl::GetColorControls** method returns the current color control settings associated with the specified overlay or primary surface. The dwFlags member of the **DDCOLORCONTROL** structure indicate which of the color control options are supported.

```
HRESULT GetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl

Address of the **DDCOLORCONTROL** structure that will receive the current control settings of the specified surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

IDirectDrawColorControl::SetColorControls

The **IDirectDrawColorControl::SetColorControls** method sets the color control settings associated with the specified overlay or primary surface.

```
HRESULT SetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl

Address of the **DDCOLORCONTROL** structure containing the new values to be applied to the specified surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

IDirectDrawPalette

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

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

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

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

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAWPALETTE data type to declare a variable that contains a pointer to an **IDirectDrawPalette** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawPalette    FAR *LPDIRECTDRAWPALETTE;
```

IDirectDrawPalette::GetCaps

The **IDirectDrawPalette::GetCaps** method retrieves the capabilities of this palette object.

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

Parameters

lpdwCaps

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

DDPCAPS_1BIT
DDPCAPS_2BIT
DDPCAPS_4BIT
DDPCAPS_8BIT
DDPCAPS_8BITENTRIES
DDPCAPS_ALLOW256
DDPCAPS_PRIMARYSURFACE
DDPCAPS_PRIMARYSURFACELEFT
DDPCAPS_VSYNC

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

IDirectDrawPalette::GetEntries

The **IDirectDrawPalette::GetEntries** method queries palette values from a DirectDrawPalette object.

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

Parameters

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

dwBase
Start of the entries that should be retrieved sequentially.

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPALETTIZED

See Also

IDirectDrawPalette::SetEntries

IDirectDrawPalette::Initialize

The **IDirectDrawPalette::Initialize** method initializes the DirectDrawPalette object.

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

Parameters

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

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

Return Values

This method returns DDERR_ALREADYINITIALIZED.

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

See Also

IUnknown::AddRef, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawPalette::SetEntries

The **IDirectDrawPalette::SetEntries** method changes entries in a DirectDrawPalette object immediately.

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

Parameters

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

dwStartingEntry
First entry to be set.

dwCount
Number of palette entries to be changed.

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOPALETTEATTACHED
DDERR_NOTPALETTIZED
DDERR_UNSUPPORTED

See Also

IDirectDrawPalette::GetEntries, **IDirectDrawSurface3::SetPalette**

IDirectDrawSurface3

Applications use the methods of the **IDirectDrawSurface3** interface to create DirectDrawSurface objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see Surfaces.

The methods of the **IDirectDrawSurface3** interface can be organized into the following groups:

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

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

The **IDirectDrawSurface3** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAWSURFACE, LPDIRECTDRAWDURFACE2, or LPDIRECTDRAWSURFACE3 data types to declare variables that point to an **IDirectDrawSurface**, **IDirectDrawSurface2**, or **IDirectDrawSurface3** interface. The Ddraw.h header file declares these data types with the following code:

```
typedef struct IDirectDrawSurface     FAR *LPDIRECTDRAWSURFACE;
typedef struct IDirectDrawSurface2    FAR *LPDIRECTDRAWSURFACE2;
typedef struct IDirectDrawSurface3    FAR *LPDIRECTDRAWSURFACE3;
```

IDirectDrawSurface3::AddAttachedSurface

The **IDirectDrawSurface3::AddAttachedSurface** method attaches a surface to another surface.

```
HRESULT AddAttachedSurface(
  LPDIRECTDRAWSURFACE3 lpDDSAttachedSurface
);
```

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANNOTATTACHSURFACE
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACEALREADYATTACHED
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

Remarks

Possible attachments include z-buffers, alpha channels, and back buffers. Some attachments automatically break other attachments. For example, the 3-D z-buffer can be attached only to one back buffer at a time. Attachment is not bidirectional, and a surface cannot be attached to itself. Emulated surfaces (in system memory) cannot be attached to nonemulated surfaces. Unless one surface is a texture map, the two attached surfaces must be the same size. A flipping surface cannot be attached to another flipping surface of the same type; however, attaching two surfaces of different types is allowed. For example, a flipping z-buffer can be attached to a regular flipping surface. If a nonflipping surface is attached to another nonflipping surface of the same type, the two surfaces will become a flipping chain. If a nonflipping surface is attached to a flipping surface, it becomes part of the existing flipping chain. Additional surfaces can be added to this chain, and each call of the **IDirectDrawSurface3::Flip** method will advance one step through the surfaces.

See Also

IDirectDrawSurface3::DeleteAttachedSurface, **IDirectDrawSurface3::EnumAttachedSurfaces**

IDirectDrawSurface3::AddOverlayDirtyRect

This method is not currently implemented.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_UNSUPPORTED

See Also

IDirectDrawSurface3::UpdateOverlayDisplay

IDirectDrawSurface3::Blt

The **IDirectDrawSurface3::Blt** method performs a bit block transfer. This method (as well as the **IDirectDrawSurface3** version) does not currently support z-buffering or alpha blending (see alpha channel) during blit operations.

```
HRESULT Blt(
  LPRECT lpDestRect,
  LPDIRECTDRAWSURFACE3 lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwFlags,
  LPDDBLTFX lpDDBltFx
);
```

Parameters

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. If this parameter is NULL, the entire destination surface will be used.

lpDDSrcSurface

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

lpSrcRect

Address of a **RECT** structure that defines the upper-left and lower-right points of the rectangle on the source surface to be blitted from. If this parameter is NULL, the entire source surface will be used.

dwFlags

DDBLT_ALPHADEST

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

DDBLT_ALPHADESTCONSTOVERRIDE

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

DDBLT_ALPHADESTNEG

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

DDBLT_ALPHADESTSURFACEOVERRIDE

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

DDBLT_ALPHAEDGEBLEND

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

DDBLT_ALPHASRC

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

DDBLT_ALPHASRCCONSTOVERRIDE

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

DDBLT_ALPHASRCNEG

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

DDBLT_ALPHASRCSURFACEOVERRIDE

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

DDBLT_ASYNC

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

DDBLT_COLORFILL

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

DDBLT_DDFX

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

DDBLT_DDROPS

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

DDBLT_DEPTHFILL

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

DDBLT_KEYDEST

Uses the color key associated with the destination surface.

DDBLT_KEYDESTOVERRIDE

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

DDBLT_KEYSRC

Uses the color key associated with the source surface.

DDBLT_KEYSRCOVERRIDE

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

DDBLT_ROP

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

DDBLT_ROTATIONANGLE

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

DDBLT_WAIT

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

another error occurs.

DDBLT_ZBUFFER

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

DDBLT_ZBUFFERDESTCONSTOVERRIDE

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

DDBLT_ZBUFFERDESTOVERRIDE

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

DDBLT_ZBUFFERSRCCONSTOVERRIDE

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

DDBLT_ZBUFFERSRCOVERRIDE

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

lpDDBltFx

Address of the **DDBLTFX** structure.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

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

Remarks

This method is capable of synchronous or asynchronous blits (the default behavior), either display memory to display memory, display memory to system memory, system memory to display memory, or system memory to system memory. The blits can be performed by using 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, **IDirectDrawSurface3::Blt** returns immediately with an error if the blitter is busy and the blit could not be set up. Specify the DDBLT_WAIT flag to request a synchronous blit. When you include the DDBLT_WAIT flag, the method waits until the blit can be set up or another error occurs before it returns.

Note that **RECT** structures are defined so that the **right** and **bottom** members are exclusive—therefore, **right** - **left** equals the width of the rectangle, not one less than the width.

IDirectDrawSurface3::BltBatch

The **IDirectDrawSurface3::BltBatch** method performs a sequence of **IDirectDrawSurface3::Blt** operations from several sources to a single destination. This method is currently only a stub; it has not yet been implemented.

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

Parameters

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

dwCount
Number of blit operations to be performed.

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

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

IDirectDrawSurface3::BltFast

The **IDirectDrawSurface3::BltFast** method performs a source copy blit or transparent blit by using a source color key or destination color key.

```
HRESULT BltFast(
  DWORD dwX,
  DWORD dwY,
  LPDIRECTDRAWSURFACE3 lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwTrans
);
```

Parameters

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

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

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

dwTrans
Type of transfer.

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

DDBLTFAST_NOCOLORKEY
Specifies a normal copy blit with no transparency.

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_EXCEPTION
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOBLTHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

Remarks

This method always attempts an asynchronous blit if it is supported by the hardware.

This method works only on display memory surfaces and cannot clip when blitting. If you use this method on a surface with an attached clipper, the call will fail and the method will return DDERR_UNSUPPORTED.

The software implementation of **IDirectDrawSurface3::BltFast** is 10 percent faster than the **IDirectDrawSurface3::Blt** method. However, there is no speed difference between the two if display hardware is being used.

Typically, **IDirectDrawSurface3::BltFast** returns immediately with an error if the blitter is busy and the blit cannot be set up. You can use the DDBLTFAST_WAIT flag, however, if you want this method to not return until either the blit can be set up or another error occurs.

IDirectDrawSurface3::DeleteAttachedSurface

The **IDirectDrawSurface3::DeletAttachedSurface** method detaches two attached surfaces. The detached surface is not released.

```
HRESULT DeleteAttachedSurface(
  DWORD dwFlags,
  LPDIRECTDRAWSURFACE3 lpDDSAttachedSurface
);
```

Parameters

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

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANNOTDETACHSURFACE
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_SURFACENOTATTACHED

Remarks

Implicit attachments, those formed by DirectDraw rather than the **IDirectDrawSurface3::AddAttachedSurface** method, cannot be detached. Detaching surfaces from a flipping chain can alter other surfaces in the chain. If a front buffer is detached from a flipping chain, the next surface in the chain becomes the front buffer, and the following surface becomes the back buffer. If a back buffer is detached from a chain, the following surface becomes a back buffer. If a plain surface is detached from a chain, the chain simply becomes shorter. If a flipping chain has only two surfaces and they are detached, the chain is destroyed and both surfaces return to their previous designations.

See Also

IDirectDrawSurface3::Flip

IDirectDrawSurface3::EnumAttachedSurfaces

The **IDirectDrawSurface3::EnumAttachedSurfaces** method enumerates all the surfaces attached to a given surface.

```
HRESULT EnumAttachedSurfaces(
  LPVOID lpContext,
  LPDDENUMSURFACESCALLBACK lpEnumSurfacesCallback
);
```

Parameters

lpContext

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

lpEnumSurfacesCallback

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_SURFACELOST

IDirectDrawSurface3::EnumOverlayZOrders

The **IDirectDrawSurface3::EnumOverlayZOrders** method enumerates the overlay surfaces on the specified destination. The overlays can be enumerated in front-to-back or back-to-front order.

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

Parameters

dwFlags

One of the following flags:

DDENUMOVERLAYZ_BACKTOFRONT

Enumerates overlays back to front.

DDENUMOVERLAYZ_FRONTTOBACK

Enumerates overlays front to back.

lpContext

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

lpfnCallback

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

IDirectDrawSurface3::Flip

The **IDirectDrawSurface3::Flip** method makes the surface memory associated with the DDSCAPS_BACKBUFFER surface become associated with the front-buffer surface.

```
HRESULT Flip(
  LPDIRECTDRAWSURFACE3 lpDDSurfaceTargetOverride,
  DWORD dwFlags
);
```

Parameters

lpDDSurfaceTargetOverride

Address of another surface in the flipping chain that will be flipped to. The specified surface must be a member of the flipping chain. The default for this parameter is NULL, in which case DirectDraw cycles through the buffers in the order they are attached to each other.

dwFlags

Flags specifying flip options.

DDFLIP_EVEN

For use only when displaying video in an overlay surface. The new surface contains data from the even field of a video signal. This flag cannot be used with the DDFLIP_ODD flag.

DDFLIP_ODD

For use only when displaying video in an overlay surface. The new surface contains data from the odd field of a video signal. This flag cannot be used with the DDFLIP_EVEN flag.

DDFLIP_WAIT

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOFLIPHW
DDERR_NOTFLIPPABLE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method can be called only for a surface that has the DDSCAPS_FLIP and DDSCAPS_FRONTBUFFER capabilities. The display memory previously associated with the front buffer is associated with the back buffer.

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 **IDirectDrawSurface3::Flip** method will always be synchronized with the vertical blank. If the surface has been assigned to a video port, this method updates the visible overlay surface and the video port's target surface.

For more information, see Flipping Surfaces.

See Also

IDirectDrawSurface3::GetFlipStatus

IDirectDrawSurface3::GetAttachedSurface

The **IDirectDrawSurface3::GetAttachedSurface** method obtains the attached surface that has the specified capabilities.

```
HRESULT GetAttachedSurface(
  LPDDSCAPS lpDDSCaps,
  LPDIRECTDRAWSURFACE3 FAR *lplpDDAttachedSurface
);
```

Parameters

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

lplpDDAttachedSurface
Address of a variable that will contain a pointer to the retrieved surface's **IDirectDrawSurface3** interface. The retrieved surface is the one that matches the description according to the *lpDDSCaps* parameter.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTFOUND
DDERR_SURFACELOST

Remarks

Attachments are used to connect multiple DirectDrawSurface objects into complex structures, like the ones needed to support 3-D page flipping with z-buffers. This method fails if more than one surface is attached that matches the capabilities requested. In this case, the application must use the **IDirectDrawSurface3::EnumAttachedSurfaces** method to obtain the attached surfaces.

IDirectDrawSurface3::GetBltStatus

The **IDirectDrawSurface3::GetBltStatus** method obtains the blitter status.

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

Parameters

dwFlags

One of the following flags:

DDGBS_CANBLT

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

DDGBS_ISBLTDONE

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

Return Values

If the method succeeds, that means a blitter is present, the return value is DD_OK.

If the method fails, the return value is DDERR_WASSTILLDRAWING if the blitter is busy, DDERR_NOBLTHW if there is no blitter, or one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOBLTHW
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

IDirectDrawSurface3::GetCaps

The **IDirectDrawSurface3::GetCaps** method retrieves the capabilities of the surface. These capabilities are not necessarily related to the capabilities of the display device.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

IDirectDrawSurface3::GetClipper

The **IDirectDrawSurface3::GetClipper** method retrieves the DirectDrawClipper object associated with this surface.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCLIPPERATTACHED

See Also

IDirectDrawSurface3::SetClipper

IDirectDrawSurface3::GetColorKey

The **IDirectDrawSurface3::GetColorKey** method retrieves the color key value for the DirectDrawSurface object.

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

Parameters

dwFlags

Determines which color key is requested.

DDCKEY_DESTBLT

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

DDCKEY_DESTOVERLAY

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

DDCKEY_SRCBLT

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

DDCKEY_SRCOVERLAY

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

lpDDColorKey

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOCOLORKEY
DDERR_NOCOLORKEYHW
DDERR_SURFACELOST
DDERR_UNSUPPORTED

See Also

IDirectDrawSurface3::SetColorKey

IDirectDrawSurface3::GetDC

The **IDirectDrawSurface3::GetDC** method creates a GDI-compatible handle of a device context for the surface.

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

Parameters

lphDC
Address for the returned handle to a device context.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_DCALREADYCREATED
DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_WASSTILLDRAWING

Remarks

This method uses an internal version of the **IDirectDrawSurface3::Lock** method to lock the surface. The surface remains locked until the **IDirectDrawSurface3::ReleaseDC** method is called.

See Also

IDirectDrawSurface3::Lock

IDirectDrawSurface3::GetDDInterface

The **IDirectDrawSurface3::GetDDInterface** method retrieves an interface to the DirectDraw object that was used to create the surface.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

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

IDirectDrawSurface3::GetFlipStatus

The **IDirectDrawSurface3::GetFlipStatus** method indicates whether the surface has finished its flipping process.

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

Parameters

dwFlags

One of the following flags:

DDGFS_CANFLIP

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

DDGFS_ISFLIPDONE

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is DDERR_WASSTILLDRAWING if the surface has not finished its flipping process, or one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_UNSUPPORTED

See Also

IDirectDrawSurface3::Flip

IDirectDrawSurface3::GetOverlayPosition

Given a visible, active overlay surface (DDSCAPS_OVERLAY flag set), the **IDirectDrawSurface3::GetOverlayPosition** method returns the display coordinates of the surface.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPOSITION
DDERR_NOOVERLAYDEST
DDERR_NOTAOVERLAYSURFACE
DDERR_OVERLAYNOTVISIBLE
DDERR_SURFACELOST

See Also

IDirectDrawSurface3::SetOverlayPosition, **IDirectDrawSurface3::UpdateOverlay**

IDirectDrawSurface3::GetPalette

The **IDirectDrawSurface3::GetPalette** method retrieves the DirectDrawPalette structure associated with this surface and increments the reference count of the returned palette.

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

Parameters

lplpDDPalette

Address of a pointer to a DirectDrawPalette structure associated with this surface. This parameter will be set to NULL if no DirectDrawPalette structure is associated with this surface.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_SURFACELOST
DDERR_UNSUPPORTED

See Also

IDirectDrawSurface3::SetPalette

IDirectDrawSurface3::GetPixelFormat

The **IDirectDrawSurface3::GetPixelFormat** method retrieves the color and pixel format of the surface.

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

Parameters

lpDDPixelFormat

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_INVALIDSURFACETYPE

IDirectDrawSurface3::GetSurfaceDesc

The **IDirectDrawSurface3::GetSurfaceDesc** method retrieves a **DDSURFACEDESC** structure that describes the surface in its current condition.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

DDSURFACEDESC

IDirectDrawSurface3::Initialize

The **IDirectDrawSurface3::Initialize** method initializes a DirectDrawSurface object.

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

Parameters

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

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

Return Values

The method returns DDERR_ALREADYINITIALIZED.

Remarks

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

See Also

IUnknown::AddRef, **IUnknown::QueryInterface**, **IUnknown::Release**

IDirectDrawSurface3::IsLost

The **IDirectDrawSurface3::IsLost** method determines if the surface memory associated with a DirectDrawSurface object has been freed.

HRESULT IsLost();

Return Values

If the method succeeds, the return value is DD_OK because the memory has not been freed.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

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

Remarks

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

See Also

IDirectDrawSurface3::Restore

IDirectDrawSurface3::Lock

The **IDirectDrawSurface3::Lock** method obtains a pointer to the surface memory.

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

Parameters

lpDestRect

Address of a **RECT** structure that identifies the region of surface that is being locked. If NULL, the entire surface will be locked.

lpDDSurfaceDesc

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

dwFlags

DDLOCK_EVENT

This flag is not currently implemented.

DDLOCK_NOSYSLOCK

If possible, do not take the Win16Lock. This flag is ignored when locking the primary surface.

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

If a lock cannot be obtained because a blit operation is in progress, the method retries until a lock is obtained or another error occurs, such as DDERR_SURFACEBUSY.

DDLOCK_WRITEONLY

Indicates that the surface being locked will only be written to.

hEvent

This parameter is not used and must be set to NULL.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_OUTOFMEMORY

DDERR_SURFACEBUSY
DDERR_SURFACELOST
DDERR_WASSTILLDRAWING

Remarks

For more information on using this method, see Accessing the Frame-Buffer Directly.

After retrieving a surface memory pointer, you can access the surface memory until a corresponding **IDirectDrawSurface3::Unlock** method is called. When the surface is unlocked, the pointer to the surface memory is invalid.

Do not call DirectDraw blit functions to blit from a locked region of a surface. If you do, the blit returns either DDERR_SURFACEBUSY or DDERR_LOCKEDSURFACES. Additionally, GDI blit functions will silently fail when used on a locked video memory surface.

This method often causes DirectDraw to hold the Win16Lock until you call the **IDirectDrawSurface3::Unlock** method. GUI debuggers cannot operate while the Win16Lock is held.

See Also

IDirectDrawSurface3::Unlock, **IDirectDrawSurface3::GetDC**, **IDirectDrawSurface3::ReleaseDC**

IDirectDrawSurface3::PageLock

The **IDirectDrawSurface3::PageLock** method prevents a system-memory surface from being paged out while a blit operation using direct memory access (DMA) transfers to or from system memory is in progress.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTPAGELOCK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

Remarks

You must call this method to make use of DMA support. If you do not, the blit occurs using software emulation. For more information, see Using DMA.

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

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface3::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface3::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

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

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

See Also

IDirectDrawSurface3::PageUnlock

IDirectDrawSurface3::PageUnlock

The **IDirectDrawSurface3::PageUnlock** method unlocks a system-memory surface, allowing it to be paged out.

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

Parameters

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_CANTPAGEUNLOCK
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOTPAGELOCKED
DDERR_SURFACELOST

Remarks

A lock count is maintained for each surface and is incremented each time **IDirectDrawSurface3::PageLock** is called for that surface. The count is decremented when **IDirectDrawSurface3::PageUnlock** is called. When the count reaches 0, the memory is unlocked and can then be paged by the operating system.

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

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

See Also

IDirectDrawSurface3::PageLock

IDirectDrawSurface3::ReleaseDC

The **IDirectDrawSurface3::ReleaseDC** method releases the handle of a device context previously obtained by using the **IDirectDrawSurface3::GetDC** method.

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

Parameters

hDC
Handle to a device context previously obtained by **IDirectDrawSurface3::GetDC**.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST
DDERR_UNSUPPORTED

Remarks

This method also unlocks the surface previously locked when the **IDirectDrawSurface3::GetDC** method was called.

See Also

IDirectDrawSurface3::GetDC

IDirectDrawSurface3::Restore

The **IDirectDrawSurface3::Restore** method restores a surface that has been lost. This occurs when the surface memory associated with the DirectDrawSurface object has been freed.

HRESULT Restore();

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_IMPLICITLYCREATED
DDERR_INCOMPATIBLEPRIMARY
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_NOEXCLUSIVEMODE
DDERR_OUTOFMEMORY
DDERR_UNSUPPORTED
DDERR_WRONGMODE

Remarks

This method restores the memory allocated for a surface, but not the contents of that memory. The application is responsible for restoring the contents to reestablish any lost images.

Surfaces can be lost because the mode of the display card was changed or because an application received exclusive access to the display card and freed all of the surface memory currently allocated on the card. When a DirectDrawSurface object loses its surface memory, many methods will return DDERR_SURFACELOST and perform no other function. The **IDirectDrawSurface3::Restore** method will reallocate surface memory and reattach it to the DirectDrawSurface object.

A single call to this method will restore a DirectDrawSurface object's associated implicit surfaces (back buffers, and so on). An attempt to restore an implicitly created surface will result in an error. **IDirectDrawSurface3::Restore** will not work across explicit attachments created by using the **IDirectDrawSurface3::AddAttachedSurface** method – each of these surfaces must be restored individually.

See Also

IDirectDrawSurface3::IsLost, **IDirectDrawSurface3::AddAttachedSurface**

IDirectDrawSurface3::SetClipper

The **IDirectDrawSurface3::SetClipper** method attaches a DirectDrawClipper object to a DirectDrawSurface object.

```
HRESULT SetClipper(
  LPDIRECTDRAWCLIPPER lpDDClipper
);
```

Parameters

lpDDClipper

Address of the DirectDrawClipper structure representing the DirectDrawClipper object that will be attached to the DirectDrawSurface object. If this parameter is NULL, the current DirectDrawClipper object will be detached.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_NOCLIPPERATTACHED

Remarks

This method is primarily used by surfaces that are being overlaid on or blitted to the primary surface. However, it can be used on any surface. After a DirectDrawClipper object has been attached and a clip list is associated with it, the DirectDrawClipper object will be used for the **IDirectDrawSurface3::Blt**, **IDirectDrawSurface3::BltBatch**, and **IDirectDrawSurface3::UpdateOverlay** operations involving the parent DirectDrawSurface object. This method can also detach a DirectDrawSurface object's current DirectDrawClipper object.

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

See Also

IDirectDrawSurface3::GetClipper

IDirectDrawSurface3::SetColorKey

The **IDirectDrawSurface3::SetColorKey** method sets the color key value for the DirectDrawSurface object if the hardware supports color keys on a per surface basis.

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

Parameters

dwFlags

Determines which color key is requested.

DDCKEY_COLORSPACE

Set if the structure contains a color space. Not set if the structure contains a single color key.

DDCKEY_DESTBLT

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

DDCKEY_DESTOVERLAY

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

DDCKEY_SRCBLT

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

DDCKEY_SRCOVERLAY

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

lpDDColorKey

Address of the **DDCOLORKEY** structure that contains the new color key values for the DirectDrawSurface object.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_INVALIDSURFACETYPE

DDERR_NOOVERLAYHW

DDERR_NOTAOVERLAYSURFACE

DDERR_SURFACELOST

DDERR_UNSUPPORTED

DDERR_WASSTILLDRAWING

Remarks

For transparent blits and overlays, you should set destination color on the destination surface and source color on the source surface.

See Also

IDirectDrawSurface3::GetColorKey

IDirectDrawSurface3::SetOverlayPosition

The **IDirectDrawSurface3::SetOverlayPosition** method changes the display coordinates of an overlay surface.

```
HRESULT SetOverlayPosition(
  LONG lX,
  LONG lY
);
```

Parameters

lX and *lY*
New x- and y-display coordinates.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPOSITION
DDERR_NOOVERLAYDEST
DDERR_NOTAOVERLAYSURFACE
DDERR_OVERLAYNOTVISIBLE
DDERR_SURFACELOST
DDERR_UNSUPPORTED

See Also

IDirectDrawSurface3::GetOverlayPosition, **IDirectDrawSurface3::UpdateOverlay**

IDirectDrawSurface3::SetPalette

The **IDirectDrawSurface3::SetPalette** method attaches the specified DirectDrawPalette object to a surface. The surface uses this palette for all subsequent operations. The palette change takes place immediately, without regard to refresh timing.

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

Parameters

lpDDPalette
Address of the DirectDrawPalette structure that this surface should use for future operations.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDPIXELFORMAT
DDERR_INVALIDSURFACETYPE
DDERR_NOEXCLUSIVEMODE
DDERR_NOPALETTEATTACHED
DDERR_NOPALETTEHW
DDERR_NOT8BITCOLOR
DDERR_SURFACELOST
DDERR_UNSUPPORTED

Remarks

If this method is called several times consecutively on the same surface for the same palette, the reference count for the palette is incremented only once. Subsequent calls do not affect the palette's reference count.

See Also

IDirectDrawSurface3::GetPalette, **IDirectDraw2::CreatePalette**

IDirectDrawSurface3::SetSurfaceDesc

The **IDirectDrawSurface3::SetSurfaceDesc** method sets the characteristics of an existing surface. This method is new with the **IDirectDrawSurface3** interface.

```
HRESULT IDirectDrawSurface3::SetSurfaceDesc(
  LPDDSURFACEDESC lpddsd,
  DWORD dwFlags
);
```

Parameters

lpddsd
Address of a **DDSURFACEDESC** structure that contains the new surface characteristics.

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

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDPARAMS
DDERR_INVALIDOBJECT
DDERR_SURFACELOST
DDERR_SURFACEBUSY
DDERR_INVALIDSURFACETYPE
DDERR_INVALIDPIXELFORMAT
DDERR_INVALIDCAPS
DDERR_UNSUPPORTED
DDERR_GENERIC

Remarks

Currently, this method can only be used to set the surface data and pixel format used by an explicit system memory surface. This is useful as it allows a surface to use data from a previously allocated buffer without copying. The new surface memory is allocated by the client application and, as such, the client application must also deallocate it. For more information about how this method is used, see Updating Surface Characteristics.

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

IDirectDrawSurface3::Unlock

The **IDirectDrawSurface3::Unlock** method notifies DirectDraw that the direct surface manipulations are complete.

```
HRESULT Unlock(
  LPVOID lpSurfaceData
);
```

Parameters

lpSurfaceData

Address of the surface to be unlocked, as retrieved by the **IDirectDrawSurface3::Lock** method. This parameter can be NULL only if the entire surface was locked by passing NULL in the *lpDestRect* parameter of the corresponding call to the **IDirectDrawSurface3::Lock** method.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_GENERIC
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_NOTLOCKED
DDERR_SURFACELOST

Remarks

Because it is possible to call **IDirectDrawSurface3::Lock** multiple times for the same surface with different destination rectangles, the pointer in *lpSurfaceData* links the calls to the **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods.

See Also

IDirectDrawSurface3::Lock

IDirectDrawSurface3::UpdateOverlay

The **IDirectDrawSurface3::UpdateOverlay** method repositions or modifies the visual attributes of an overlay surface. These surfaces must have the DDSCAPS_OVERLAY value set.

```
HRESULT UpdateOverlay(
  LPRECT lpSrcRect,
  LPDIRECTDRAWSURFACE3 lpDDDestSurface,
  LPRECT lpDestRect,
  DWORD dwFlags,
  LPDDOVERLAYFX lpDDOverlayFx
);
```

Parameters

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. This parameter can be NULL when hiding an overlay or to indicate that the entire overlay surface is to be used and that the overlay surface conforms to any boundary and size alignment restrictions imposed by the device driver.

lpDDDestSurface

Address of the DirectDraw surface that is being overlaid.

lpDestRect

Address of a **RECT** structure that defines the x, y, width, and height of the region on the destination surface that the overlay should be moved to. This parameter can be NULL when hiding the overlay.

dwFlags

DDOVER_ADDDIRTYRECT

Adds a dirty rectangle to an emulated overlaid surface.

DDOVER_ALPHADEST

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

DDOVER_ALPHADESTCONSTOVERRIDE

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

DDOVER_ALPHADESTNEG

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

DDOVER_ALPHADESTSURFACEOVERRIDE

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

DDOVER_ALPHAEDGEBLEND

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

DDOVER_ALPHASRC

Uses either the alpha information in pixel format or the alpha channel surface attached to the source surface as the source

alpha channel for this overlay.

DDOVER_ALPHASRCCONSTOVERRIDE

Uses the **dwAlphaSrcConst** member of the **DDOVERLAYFX** structure as the source alpha channel for this overlay.

DDOVER_ALPHASRCNEG

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

DDOVER_ALPHASRCSURFACEOVERRIDE

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

DDOVER_AUTOFLIP

Automatically flip to the next surface in the flip chain each time a video port VSYNC occurs.

DDOVER_BOB

Display each field individually of the interlaced video stream without causing any artifacts.

DDOVER_DDFX

Uses the overlay FX flags in the *lpDDOverlayFx* parameter to define special overlay effects.

DDOVER_HIDE

Turns this overlay off.

DDOVER_KEYDEST

Uses the color key associated with the destination surface.

DDOVER_KEYDESTOVERRIDE

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

DDOVER_KEYSRC

Uses the color key associated with the source surface.

DDOVER_KEYSRCOVERRIDE

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

DDOVER_OVERRIDEBOBWEAVE

Indicates that bob/weave decisions should not be overridden by other interfaces.

DDOVER_INTERLEAVED

Indicates that the surface memory is composed of interleaved fields.

DDOVER_SHOW

Turns this overlay on.

lpDDOverlayFx

Address of a **DDOVERLAYFX** structure that describes the effects to be used. This parameter can be NULL if the DDOVER_DDFX flag is not specified.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_DEVICEDOESNTOWNSURFACE
DDERR_GENERIC
DDERR_HEIGHTALIGN
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDRECT
DDERR_INVALIDSURFACETYPE
DDERR_NOSTRETCHHW
DDERR_NOTAOVERLAYSURFACE
DDERR_OUTOFCAPS
DDERR_SURFACELOST
DDERR_UNSUPPORTED
DDERR_XALIGN

IDirectDrawSurface3::UpdateOverlayDisplay

The **IDirectDrawSurface3::UpdateOverlayDisplay** method 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.

```
HRESULT UpdateOverlayDisplay(
  DWORD dwFlags
);
```

Parameters

dwFlags

Type of update to perform. One of the following flags:

DDOVER_REFRESHDIRTYRECTS

Updates the overlay display using the list of dirty rectangles previously constructed for this destination. This clears the dirty rectangle list.

DDOVER_REFRESHALL

Ignores the dirty rectangle list and updates the overlay display completely. This clears the dirty rectangle list.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_INVALIDSURFACETYPE
DDERR_UNSUPPORTED

Remarks

This method is not currently implemented.

See Also

IDirectDrawSurface3::AddOverlayDirtyRect

IDirectDrawSurface3::UpdateOverlayZOrder

The **IDirectDrawSurface3::UpdateOverlayZOrder** method sets the z-order of an overlay.

```
HRESULT UpdateOverlayZOrder(
  DWORD dwFlags,
  LPDIRECTDRAWSURFACE3 lpDDSReference
);
```

Parameters

dwFlags

One of the following flags:

DDOVERZ_INSERTINBACKOF

Inserts this overlay in the overlay chain behind the reference overlay.

DDOVERZ_INSERTINFRONTOF

Inserts this overlay in the overlay chain in front of the reference overlay.

DDOVERZ_MOVEBACKWARD

Moves this overlay one position backward in the overlay chain.

DDOVERZ_MOVEFORWARD

Moves this overlay one position forward in the overlay chain.

DDOVERZ_SENDTOBACK

Moves this overlay to the back of the overlay chain.

DDOVERZ_SENDTOFRONT

Moves this overlay to the front of the overlay chain.

lpDDSReference

Address of the DirectDraw surface to be used as a relative position in the overlay chain. This parameter is needed only for DDOVERZ_INSERTINBACKOF and DDOVERZ_INSERTINFRONTOF.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_NOTAOVERLAYSURFACE

See Also

IDirectDrawSurface3::EnumOverlayZOrders

IDirectDrawVideoPort

Applications use the methods of the **IDirectDrawVideoPort** interface to channel live video data from a hardware video port to a DirectDraw surface. This section is a reference to the methods of this interface. For a conceptual overview, see Video-Ports.

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

Color controls	**GetColorControls** **SetColorControls**
Fields and Signals	**GetFieldPolarity** **GetVideoSignalStatus**
Flipping	**Flip** **SetTargetSurface**
Formats	**GetInputFormats** **GetOutputFormats**
Timing and Synchronization	**GetVideoLine** **WaitForSync**
Video control	**StartVideo** **StopVideo** **UpdateVideo**
Zoom factors	**GetBandwidthInfo**

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

AddRef
QueryInterface
Release

You can use the LPDIRECTDRAWVIDEOPORT data type to declare a variable that contains a pointer to an **IDirectDrawVideoPort** interface. The Dvp.h header file declares the LPDIRECTDRAWVIDEOPORT with the following code:

```
typedef struct IDirectDrawVideoPort    FAR *LPDIRECTDRAWVIDEOPORT;
```

IDirectDrawVideoPort::Flip

The **IDirectDrawVideoPort::Flip** method instructs the DirectDrawVideoPort object to write the next frame of video to a new surface.

```
HRESULT Flip(
  LPDIRECTDRAWSURFACE lpDDSurface,
  DWORD dwFlags
);
```

Parameters

lpDDSurface

Address of the DirectDrawSurface object that will receive the next frame of video.

dwFlags

Flip options flags. This parameter can be one of the following values.

DDVPFLIP_VIDEO

The specified surface is to receive the normal video data.

DDVPFLIP_VBI

The specified surface is to receive only the data within the vertical blanking interval.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method can be used to prevent tearing. Calls to **IDirectDrawVideoPort::Flip** are asynchronous—the actual flip operation will always be synchronized with the vertical blank of the video signal.

IDirectDrawVideoPort::GetBandwidthInfo

The **IDirectDrawVideoPort::GetBandwidthInfo** method retrieves the minimum required overlay zoom factors and device limitations of a video port that uses the provided output pixel format.

HRESULT GetBandwidthInfo(

```
  LPDDPIXELFORMAT lpddpfFormat,
  DWORD dwWidth,
  DWORD dwHeight,
  DWORD dwFlags,
  LPDDVIDEOPORTBANDWIDTH lpBandwidth
);
```

Parameters

lpddpfFormat

Address of a **DDPIXELFORMAT** structure that describes the output pixel format for which bandwidth information will be retrieved.

dwWidth and *dwHeight*

Dimensions of an overlay or video data. These interpretation of these parameters depends on the value specified in the *dwFlags* parameter.

dwFlags

Flags indicating how the method is to interpret the *dwWidth* and *dwHeight* parameters. This parameter can be one of the following values.

DDVPB_OVERLAY

The *dwWidth* and *dwHeight* parameters indicate the size of the source overlay surface. Use this flag when the video port is dependent on the overlay source size.

DDVPB_TYPE

The *dwWidth* and *dwHeight* parameters are not set. The method will retrieve the device's dependency type in the **dwCaps** member of the associated **DDVIDEOPORTBANDWIDTH** structure. Use this flag when you call this method the first time.

DDVPB_VIDEOPORT

The *dwWidth* and *dwHeight* parameters indicate the prescale size of the of the video data that the video port writes to the frame buffer. Use this flag when the video port is dependent on the overlay zoom factor.

lpBandwidth

Address of a **DDVIDEOPORTBANDWIDTH** structure that will be filled with the retrieved bandwidth and device dependency information.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

Remarks

This method will usually be called twice. When you make the first call, specify the DDVPB_TYPE flag in the *dwFlags* parameter to retrieve information about device's overlay dependency type. Subsequent calls using the DDVPB_VIDEOPORT or DDVPB_OVERLAY flags must be interpreted considering the device's dependency type.

IDirectDrawVideoPort::GetColorControls

The **IDirectDrawVideoPort::GetColorControls** method returns the current color control settings associated with the video port.

```
HRESULT GetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl

Address of a **DDCOLORCONTROL** structure that will be filled with the current settings of the video port's color control.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_UNSUPPORTED

Remarks

The **dwFlags** member of the **DDCOLORCONTROL** structure indicate which of the color control options are supported.

IDirectDrawVideoPort::GetInputFormats

The **IDirectDrawVideoPort::GetInputFormat** method retrieves the input formats supported by the DirectDrawVideoPort object.

```
HRESULT GetInputFormats(
  LPDWORD lpNumFormats,
  LPDDPIXELFORMAT lpFormats,
  DWORD dwFlags
);
```

Parameters

lpNumFormats

Address of a variable containing the number of entries that the array at *lpFromats* can hold. If this number is less than the total number of codes, the method fills the array with as many codes as will fit, sets the value at *lpNumFormats* to indicate the total number of codes, and returns DDERR_MOREDATA.

lpFormats

Address of an array of **DDPIXELFORMAT** structures that will be filled in with the input formats supported by this DirectDrawVideoPort object. If this parameter is NULL, the method sets *lpNumFormats* to the number of supported formats and the returns DD_OK.

dwFlags

Flags specifying the part of the video signal for which formats will be enumerated. This parameter can be one of the following values.

DDVPFORMAT_VIDEO

Returns formats for the video data.

DDVPFORMAT_VBI

Returns formats for the VBI data.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_MOREDATA

Remarks

This method can also be used to return the number of formats supported. To do this, set the *lpFormats* parameter to NULL. When the method returns, the variable at *lpNumFormats* contains the total number of supported input formats.

IDirectDrawVideoPort::GetOutputFormats

The **IDirectDrawVideoPort::GetOutputFormats** method retrieves a list of output formats that the DirectDrawVideoPort object supports for a specified input format.

HRESULT GetOutputFormats(

```
  LPDDPIXELFORMAT lpInputFormat,
  LPDWORD lpNumFormats,
  LPDDPIXELFORMAT lpFormats,
  DWORD dwFlags
);
```

Parameters

lpInputFormat

Address of a **DDPIXELFORMAT** structure that describes the input format for which conversion information is requested.

lpNumFormats

Address of a variable containing the number of entries that the array at *lpFromats* can hold. If this number is less than the total number of codes, the method fills the array with as many codes as will fit, sets the value at *lpNumFormats* to indicate the total number of codes, and returns DDERR_MOREDATA.

lpFormats

Address of an array of **DDPIXELFORMAT** structures that will be filled in with the output formats supported by this DirectDrawVideoPort object. If this parameter is NULL, the method sets *lpNumFormats* to the number of supported formats and the returns DD_OK.

dwFlags

Flags specifying the part of the video signal for which formats will be enumerated. This parameter can be one of the following values.

DDVPFORMAT_VIDEO

Returns formats for the video data.

DDVPFORMAT_VBI

Returns formats for the VBI data.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_MOREDATA

IDirectDrawVideoPort::GetFieldPolarity

The **IDirectDrawVideoPort::GetFieldPolarity** method retrieves the status of the video field.

```
HRESULT GetVideoField(
  LPBOOL lpbVideoField
);
```

Parameters

lpbVideoField

Address of a variable that will be set to indicate the current field polarity. This value is set to true if the current video field is the even field of an interlaced video signal and false otherwise.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VIDEONOTACTIVE

IDirectDrawVideoPort::GetVideoLine

The **IDirectDrawVideoPort::GetVideoLine** method retrieves the current line of video being written to the frame buffer.

```
HRESULT GetVideoLine(
  LPDWORD lpdwLine
);
```

Parameters

lpdwLine

Address of a variable that will be filled with a value indicating the video line currently being written to the frame buffer.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VERTICALBLANKINPROGRESS
DDERR_VIDEONOTACTIVE

Remarks

The value this method retrieves reflects the true video line being written, relative to the field height, before any prescaling occurs.

IDirectDrawVideoPort::GetVideoSignalStatus

The **IDirectDrawVideoPort::GetVideoSignalStatus** method retrieves the status of the video signal currently being presented to the video port.

```
HRESULT GetVideoSignalStatus(
  LPDWORD lpdwStatus
);
```

Parameters

lpdwStatus

Address of a variable that will contain a return code indicating the quality of the video signal at the video port. The value will be set to one of the following codes.

DDVPSQ_NOSIGNAL

No video signal is present at the video port.

DDVPSQ_SIGNALOK

A valid video signal is present at the video port.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT

DDERR_INVALIDPARAMS

DDERR_UNSUPPORTED

IDirectDrawVideoPort::SetColorControls

The **IDirectDrawVideoPort::SetColorControls** method sets the color control settings associated with the video port.

```
HRESULT SetColorControls(
  LPDDCOLORCONTROL lpColorControl
);
```

Parameters

lpColorControl

Address of a **DDCOLORCONTROL** structure containing the new color control settings that will be applied to the video port.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED

IDirectDrawVideoPort::SetTargetSurface

The **IDirectDrawVideoPort::SetTargetSurface** method sets the DirectDraw surface object that will receive the stream of live video data and/or the vertical blank interval data.

HRESULT SetTargetSurface(

```
  LPDIRECTDRAWSURFACE lpDDSurface,
  DWORD dwFlags
);
```

Parameters

lpDDSurface

Address of the DirectDrawSurface object that will receive the video data.

dwFlags

Value specifying the type of target surface.

DDVPTARGET_VIDEO	The specified surface should receive the normal video data and vertical interval data unless a separate surface was attached for this purpose.
DDVPTARGET_VBI	The specified surface should receive the data within the vertical blanking interval.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirectDrawVideoPort::StartVideo, **IDirectDrawVideoPort::StopVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::StartVideo

The **IDirectDrawVideoPort::StartVideo** method enables the hardware video port and starts the flow of video data into the currently specified surface.

```
HRESULT StartVideo(
  LPDDVIDEOPORTINFO lpVideoInfo
);
```

Parameters

lpVideoInfo
Address of a pointer to a **DDVIDEOPORTINFO** structure.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_SURFACELOST

See Also

IDirectDrawVideoPort::SetTargetSurface, **IDirectDrawVideoPort::StopVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::StopVideo

The **IDirectDrawVideoPort::StopVideo** method stops the flow of video port data into the frame buffer.

HRESULT StopVideo();

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is DDERR_INVALIDOBJECT.

See Also

IDirectDrawVideoPort::SetTargetSurface, I**IDirectDrawVideoPort::StartVideo**, **IDirectDrawVideoPort::UpdateVideo**

IDirectDrawVideoPort::UpdateVideo

The **IDirectDrawVideoPort::UpdateVideo** method updates parameters that govern the flow of video data from the video-port to the DirectDrawSurface object.

```
HRESULT UpdateVideo(
  LPDDVIDEOPORTINFO lpVideoInfo
);
```

Parameters

lpVideoInfo
Address of a **DDVIDEOPORTINFO** structure that describes the video transfer parameters.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirectDrawVideoPort::SetTargetSurface, **IDirectDrawVideoPort::StartVideo**, **IDirectDrawVideoPort::StopVideo**

IDirectDrawVideoPort::WaitForSync

The **IDirectDrawVideoPort::WaitForSync** method waits for VSYNC or until a given scan line is being drawn.

```
HRESULT WaitForSync(
  DWORD dwFlags,
  DWORD dwLine,
  DWORD dwTimeout
);
```

Parameters

dwFlags

Flag specifying how the method will wait for the video VSYNC or the specified line number.

DDVPWAIT_BEGINVBLANK	Return at the start of the vertical blanking interval.
DDVPWAIT_ENDVBLANK	Return at the end of the vertical blanking interval .
DDVPWAIT_LINE	Return when the video counter either reaches or passes the line specified by the *dwLine* parameter.

dwLine

The video line determining when the method should return, relative to the field height, before prescaling. This parameter is ignored if the *dwFlags* parameter is set to DDVPWAIT_BEGINVBLANK or DDVPWAIT_ENDVBLANK.

dwTimeout

Amount of time, in milliseconds, that the method will wait for the next video vertical blank before timing out. If this parameter is 0, the method waits 3 times the value specified in the **dwMicrosecondsPerField** member of the **DDVIDEOPORTDESC**.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following error values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_UNSUPPORTED
DDERR_VIDEONOTACTIVE
DDERR_WASSTILLDRAWING

Remarks

This method helps the caller synchronize with the video vertical blank interval or with an arbitrary line of video data. The method blocks the calling thread until either the video VSYNC occurs or when the video line counter matches the specified line number.

Functions

This section contains information about the following DirectDraw global functions:

- **DirectDrawCreate**
- **DirectDrawCreateClipper**
- **DirectDrawEnumerate**

DirectDrawCreate

The **DirectDrawCreate** function creates an instance of a DirectDraw object.

```
HRESULT DirectDrawCreate(
  GUID FAR *lpGUID,
  LPDIRECTDRAW FAR *lplpDD,
  IUnknown FAR *pUnkOuter
);
```

Parameters

lpGUID

Address of the globally unique identifier (GUID) that represents the driver to be created. This can be NULL to indicate the active display driver, or you can pass one of the following flags to restrict the active display driver's behavior for debugging purposes:

DDCREATE_EMULATIONONLY

The DirectDraw object will use emulation for all features; it will not take advantage of any hardware supported features.

DDCREATE_HARDWAREONLY

The DirectDraw object will never emulate features not supported by the hardware. Attempts to call methods that require unsupported features will fail, returning DDERR_UNSUPPORTED.

lplpDD

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

pUnkOuter

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

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value may be one of the following error values:

DDERR_DIRECTDRAWALREADYCREATED

DDERR_GENERIC

DDERR_INVALIDDIRECTDRAWGUID

DDERR_INVALIDPARAMS

DDERR_NODIRECTDRAWHW

DDERR_OUTOFMEMORY

Remarks

This function attempts to initialize a DirectDraw object, and it then sets a pointer to the object if the call is successful.

On systems with multiple monitors, specifying NULL for *lpGUID* causes the DirectDraw object to run in emulation mode when the normal cooperative level is set. To make use of hardware acceleration on these systems, you must specify the device's GUID. For more information, see Working with Multiple Monitors.

DirectDrawCreateClipper

The **DirectDrawCreateClipper** function creates an instance of a DirectDrawClipper object not associated with a DirectDraw object.

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

Parameters

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

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

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

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value may be one of the following error values:

DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY

Remarks

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

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

See Also

IDirectDraw2::CreateClipper

DirectDrawEnumerate

The **DirectDrawEnumerate** function enumerates the DirectDraw objects installed on the system. The NULL GUID entry always identifies the primary display device shared with GDI.

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

Parameters

lpCallback

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

lpContext

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

Return Values

If the function succeeds, the return value is DD_OK.

If the function fails, the return value is DDERR_INVALIDPARAMS.

Remarks

On systems with multiple monitors, this method enumerates multiple display devices. For more information, see Working with Multiple Monitors.

Callback Functions

This section contains information about the following callback functions used with DirectDraw:

- **DDEnumCallback**
- **EnumModesCallback**
- **EnumSurfacesCallback**
- **EnumVideoCallback**

DDEnumCallback

The **DDEnumCallback** is an application-defined callback function for the **DirectDrawEnumerate** function.

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

Parameters

lpGUID
Address of the unique identifier of the DirectDraw object.

lpDriverDescription
Address of a string containing the driver description.

lpDriverName
Address of a string containing the driver name.

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

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the LPDDENUMCALLBACK data type to declare a variable that can contain a pointer to this callback function.

EnumModesCallback

The **EnumModesCallback** is an application-defined callback function for the **IDirectDraw2::EnumDisplayModes** method.

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

Parameters

lpDDSurfaceDesc

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

lpContext

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

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the LPDDENUMMODESCALLBACK data type to declare a variable that can contain a pointer to this callback function.

EnumSurfacesCallback

The **EnumSurfacesCallback** is an application-defined callback function for the **IDirectDrawSurface3::EnumAttachedSurfaces** and **IDirectDrawSurface3::EnumOverlayZOrders** methods.

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

Parameters

lpDDSurface
Address of the surface attached to this surface.

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

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

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

You can use the LPDDENUMSURFACESCALLBACK data type to declare a variable that can contain a pointer to this callback function.

EnumVideoCallback

The **EnumVideoCallback** is an application-defined callback procedure for the **IDDVideoPortContainer::EnumVideoPorts** method.

```
HRESULT WINAPI EnumVideoCallback(
  LPDDVIDEOPORTCAPS lpDDVideoPortCaps,
  LPVOID lpContext
);
```

Parameters

lpDDVideoPortCaps

Pointer to the **DDVIDEOPORTCAPS** structure that contains the video port information, including the ID and capabilities. This data is read-only.

lpContext

Pointer to a caller-defined structure that is passed to the member every time it is called.

Return Values

The callback function returns DDENUMRET_OK to continue the enumeration.

The callback function returns DDENUMRET_CANCEL to stop it.

Remarks

Video-port related functions cannot be called from inside the **EnumVideoCallback** function. Attempts to do so will fail, returning DDERR_CURRENTLYNOTAVAIL.

You can use the LPDDENUMVIDEOCALLBACK data type to declare a variable that can contain a pointer to this callback function.

Structures

This section contains information about the following structures used with DirectDraw:

- **DDBLTBATCH**
- **DDBLTFX**
- **DDCAPS**
- **DDCOLORCONTROL**
- **DDCOLORKEY**
- **DDOVERLAYFX**
- **DDPIXELFORMAT**
- **DDSCAPS**
- **DDSURFACEDESC**
- **DDVIDEOPORTBANDWIDTH**
- **DDVIDEOPORTCAPS**
- **DDVIDEOPORTCONNECT**
- **DDVIDEOPORTDESC**
- **DDVIDEOPORTINFO**
- **DDVIDEOPORTSTATUS**

DDBLTBATCH

The **DDBLTBATCH** structure passes blit operations to the **IDirectDrawSurface3::BltBatch** method.

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

Members

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

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

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

dwFlags
Optional control flags.

DDBLT_ALPHADEST

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

DDBLT_ALPHADESTCONSTOVERRIDE

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

DDBLT_ALPHADESTNEG

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

DDBLT_ALPHADESTSURFACEOVERRIDE

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

DDBLT_ALPHAEDGEBLEND

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

DDBLT_ALPHASRC

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

DDBLT_ALPHASRCCONSTOVERRIDE

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

DDBLT_ALPHASRCNEG

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

DDBLT_ALPHASRCSURFACEOVERRIDE

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

DDBLT_ASYNC

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

DDBLT_COLORFILL

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

DDBLT_DDFX

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

DDBLT_DDROPS

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

DDBLT_KEYDEST

Uses the color key associated with the destination surface.

DDBLT_KEYDESTOVERRIDE

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

DDBLT_KEYSRC

Uses the color key associated with the source surface.

DDBLT_KEYSRCOVERRIDE

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

DDBLT_ROP

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

DDBLT_ROTATIONANGLE

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

DDBLT_ZBUFFER

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

DDBLT_ZBUFFERDESTCONSTOVERRIDE

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

DDBLT_ZBUFFERDESTOVERRIDE

Performs a z-buffered blit using the **lpDDSZBufferDest** and

dwZBufferOpCode members of the **DDBLTFX** structure as the z-buffer and z-buffer opcode, respectively, for the destination.

DDBLT_ZBUFFERSRCCONSTOVERRIDE

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

DDBLT_ZBUFFERSRCOVERRIDE

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

lpDDBltFx

Address of a **DDBLTFX** structure specifying additional blit effects.

DDBLTFX

The **DDBLTFX** structure passes raster operations, effects, and override information to the **IDirectDrawSurface3::Blt** method. This structure is also part of the **DDBLTBATCH** structure used with the **IDirectDrawSurface3::BltBatch** method.

```
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;
        DWORD               dwFillPixel;
        LPDIRECTDRAWSURFACE lpDDSPattern;
};
DDCOLORKEY ddckDestColorkey;
DDCOLORKEY ddckSrcColorkey;
} DDBLTFX,FAR* LPDDBLTFX;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwDDFX

Type of FX operations.

DDBLTFX_ARITHSTRETCHY

Uses arithmetic stretching along the y-axis for this blit.

DDBLTFX_MIRRORLEFTRIGHT

Turns the surface on its y-axis. This blit mirrors the surface from left to right.

DDBLTFX_MIRRORUPDOWN

Turns the surface on its x-axis. This blit mirrors the surface from top to bottom.

DDBLTFX_NOTEARING

Schedules this blit to avoid tearing.

DDBLTFX_ROTATE180

Rotates the surface 180 degrees clockwise during this blit.

DDBLTFX_ROTATE270

Rotates the surface 270 degrees clockwise during this blit.

DDBLTFX_ROTATE90

Rotates the surface 90 degrees clockwise during this blit.

DDBLTFX_ZBUFFERBASEDEST

Adds the **dwZBufferBaseDest** member to each of the source z-values before comparing them with the destination z-values during this z-blit.

DDBLTFX_ZBUFFERRANGE

Uses the **dwZBufferLow** and **dwZBufferHigh** members as range values to specify limits to the bits copied from a source surface during this z-blit.

dwROP

Win32 raster operations. You can retrieve a list of supported raster operations by calling the **IDirectDraw2::GetCaps** method.

dwDDROP

DirectDraw raster operations.

dwRotationAngle

Rotation angle for the blit.

dwZBufferOpCode

Z-buffer compares.

dwZBufferLow

Low limit of a z-buffer.

dwZBufferHigh

High limit of a z-buffer.

dwZBufferBaseDest

Destination base value of a z-buffer.

dwZDestConstBitDepth

Bit depth of the destination z-constant.

dwZDestConst
Constant used as the z-buffer destination.

lpDDSZBufferDest
Surface used as the z-buffer destination.

dwZSrcConstBitDepth
Bit depth of the source z-constant.

dwZSrcConst
Constant used as the z-buffer source.

lpDDSZBufferSrc
Surface used as the z-buffer source.

dwAlphaEdgeBlendBitDepth
Bit depth of the constant for an alpha edge blend.

dwAlphaEdgeBlend
Alpha constant used for edge blending.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth
Bit depth of the destination alpha constant.

dwAlphaDestConst
Constant used as the alpha channel destination.

lpDDSAlphaDest
Surface used as the alpha channel destination.

dwAlphaSrcConstBitDepth
Bit depth of the source alpha constant.

dwAlphaSrcConst
Constant used as the alpha channel source.

lpDDSAlphaSrc
Surface used as the alpha channel source.

dwFillColor
Color used to fill a surface when DDBLT_COLORFILL is specified. This value must be a pixel appropriate to the pixel format of the destination surface. For a palettized surface it would be a palette index, and for a 16-bit RGB surface it would be a 16-bit pixel value.

dwFillDepth
Depth value for the z-buffer.

dwFillPixel
Pixel value for RGBA or RGBZ fills. Applications that use RGBZ fills should use this member, not **dwFillColor** or **dwFillDepth**.

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

The **DDCAPS** structure represents the capabilities of the hardware exposed through the DirectDraw object. This structure contains a **DDSCAPS** structure used in this context to describe what kinds of DirectDrawSurface objects can be created. It may not be possible to simultaneously create all of the surfaces described by these capabilities. This structure is used with the **IDirectDraw2::GetCaps** method.

```
typedef struct _DDCAPS {
    DWORD    dwSize;
    DWORD    dwCaps;                  // driver-specific caps
    DWORD    dwCaps2;                 // more driver-specific caps
    DWORD    dwCKeyCaps;              // color key caps
    DWORD    dwFXCaps;                // stretching and effects caps
    DWORD    dwFXAlphaCaps;           // alpha caps
    DWORD    dwPalCaps;               // palette caps
    DWORD    dwSVCaps;                // stereo vision caps
    DWORD    dwAlphaBltConstBitDepths;       // alpha bit-depth members
    DWORD    dwAlphaBltPixelBitDepths;       //  .
    DWORD    dwAlphaBltSurfaceBitDepths;     //  .
    DWORD    dwAlphaOverlayConstBitDepths;   //  .
    DWORD    dwAlphaOverlayPixelBitDepths;   //  .
    DWORD    dwAlphaOverlaySurfaceBitDepths; //  .
    DWORD    dwZBufferBitDepths;      // Z-buffer bit depth
    DWORD    dwVidMemTotal;           // total video memory
    DWORD    dwVidMemFree;            // total free video memory
    DWORD    dwMaxVisibleOverlays;    // maximum visible overlays
    DWORD    dwCurrVisibleOverlays;   // overlays currently visible
    DWORD    dwNumFourCCCodes;        // number of supported FOURCC codes
    DWORD    dwAlignBoundarySrc;      // overlay alignment restrictions
    DWORD    dwAlignSizeSrc;          //  .
    DWORD    dwAlignBoundaryDest;     //  .
    DWORD    dwAlignSizeDest;         //  .
    DWORD    dwAlignStrideAlign;      // stride alignment
    DWORD    dwRops[DD_ROP_SPACE];    // supported raster ops
    DDSCAPS  ddsCaps;                 // general surface caps
    DWORD    dwMinOverlayStretch;     // overlay stretch factors
    DWORD    dwMaxOverlayStretch;     //  .
    DWORD    dwMinLiveVideoStretch;   // obsolete
    DWORD    dwMaxLiveVideoStretch;   //  .
    DWORD    dwMinHwCodecStretch;     //  .
    DWORD    dwMaxHwCodecStretch;     //  .
    DWORD    dwReserved1;             // reserved
    DWORD    dwReserved2;             //  .
    DWORD    dwReserved3;             //  .
    DWORD    dwSVBCaps;               // system-to-video blit related caps
    DWORD    dwSVBCKeyCaps;           //  .
    DWORD    dwSVBFXCaps;             //  .
    DWORD    dwSVBRops[DD_ROP_SPACE]; //  .
    DWORD    dwSVBCaps;               // video-to-system blit related caps
    DWORD    dwVSBCKeyCaps;           //  .
    DWORD    dwVSBFXCaps;             //  .
    DWORD    dwVSBRops[DD_ROP_SPACE]; //  .
    DWORD    dwSSBCaps;               // system-to-system blit related caps
    DWORD    dwSSBCKeyCaps;           //  .
```

```
    DWORD    dwSSBCFXCaps;            //  .
    DWORD    dwSSBRops[DD_ROP_SPACE]; //  .
    DWORD    dwMaxVideoPorts;         // maximum number of live video ports
    DWORD    dwCurrVideoPorts;        // current number of live video ports
    DWORD    dwSVBCaps2;              // additional system-to-video blit
caps
    DWORD    dwNLVBCaps;              // nonlocal-to-local video memory blit
caps
    DWORD    dwNLVBCaps2;             //  .
    DWORD    dwNLVBCKeyCaps;          //  .
    DWORD    dwNLVBFXCaps;            //  .
    DWORD    dwNLVBRops[DD_ROP_SPACE];//  .
    DWORD    dwReserved4;             // reserved
    DWORD    dwReserved5;             //  .
    DWORD    dwReserved6;             //  .
} DDCAPS,FAR* LPDDCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwCaps

Driver-specific capabilities.

DDCAPS_3D

Indicates that the display hardware has 3-D acceleration.

DDCAPS_ALIGNBOUNDARYDEST

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

DDCAPS_ALIGNBOUNDARYSRC

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

DDCAPS_ALIGNSIZEDEST

Indicates that DirectDraw will support only those overlay destination rectangles whose x-axis sizes, in pixels, are **dwAlignSizeDest** multiples.

DDCAPS_ALIGNSIZESRC

Indicates that DirectDraw will support only those overlay source rectangles whose x-axis sizes, in pixels, are **dwAlignSizeSrc** multiples.

DDCAPS_ALIGNSTRIDE

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

DDCAPS_ALPHA

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

DDCAPS_BANKSWITCHED

Indicates that the display hardware is bank-switched and is

potentially very slow at random access to display memory.

DDCAPS_BLT

Indicates that display hardware is capable of blit operations.

DDCAPS_BLTCOLORFILL

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

DDCAPS_BLTDEPTHFILL

Indicates that display hardware is capable of depth filling z-buffers with a blitter.

DDCAPS_BLTFOURCC

Indicates that display hardware is capable of color-space conversions during blit operations.

DDCAPS_BLTQUEUE

Indicates that display hardware is capable of asynchronous blit operations.

DDCAPS_BLTSTRETCH

Indicates that display hardware is capable of stretching during blit operations.

DDCAPS_CANBLTSYSMEM

Indicates that display hardware is capable of blitting to or from system memory.

DDCAPS_CANCLIP

Indicates that display hardware is capable of clipping with blitting.

DDCAPS_CANCLIPSTRETCHED

Indicates that display hardware is capable of clipping while stretch blitting.

DDCAPS_COLORKEY

Supports some form of color key in either overlay or blit operations. More specific color key capability information can be found in the **dwCKeyCaps** member.

DDCAPS_COLORKEYHWASSIST

Indicates that the color key is partially hardware assisted. This means that other resources (CPU and/or video memory) might be used. If this bit is not set, full hardware support is in place.

DDCAPS_GDI

Indicates that display hardware is shared with GDI.

DDCAPS_NOHARDWARE

Indicates that there is no hardware support.

DDCAPS_OVERLAY

Indicates that display hardware supports overlays.

DDCAPS_OVERLAYCANTCLIP

Indicates that display hardware supports overlays but cannot clip them.

DDCAPS_OVERLAYFOURCC

Indicates that overlay hardware is capable of color-space conversions during overlay operations.

DDCAPS_OVERLAYSTRETCH

Indicates that overlay hardware is capable of stretching. The **dwMinOverlayStretch** and **dwMaxOverlayStretch** members contain valid data.

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 the **IDirectDrawSurface3::UpdateOverlayZOrder** method as a z-value for overlays to control their layering.

dwCaps2

More driver-specific capabilities.

DDCAPS2_AUTO FLIPOVERLAY

The overlay can be automatically flipped to the next surface in the flip chain each time a video port VSYNC occurs, allowing the video port and the overlay to double buffer the video without CPU overhead. This option is only valid when the surface is receiving data from a video port. If the video port data is non-interlaced or non-interleaved, it will flip on every VSYNC. If the data is being interleaved in memory, it will flip on every other VSYNC.

DDCAPS2_CANBOBINTERLEAVED

The overlay hardware can display each field individually of an interlaced video stream while it is interleaved in memory without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least 2X in the vertical direction.

DDCAPS2_CANBOBNONINTERLEAVED

The overlay hardware can display each field individually of an interlaced video stream while it is not interleaved in memory

without causing any artifacts that might normally occur without special hardware support. This option is only valid when the surface is receiving data from a video port and is only valid when the video is zoomed at least 2X in the vertical direction.

DDCAPS2_CANDROPZ16BIT

Sixteen-bit RGBZ values can be converted into sixteen-bit RGB values. (The system does not support eight-bit conversions.)

DDCAPS2_CANFLIPODDEVEN

The driver is capable of performing odd and even flip operations, as specified by the DDFLIP_ODD and DDFLIP_EVEN flags used with the **IDirectDrawSurface3::Flip** method.

DDCAPS2_CANSMOOTHINTERLEAVED

Overlay can display each field of interlaced data individually while it is interleaved in memory without causing jittery artifacts.

DDCAPS2_CANSMOOTHNONINTERLEAVED

Overlay can display each field of interlaced data individually while it is not interleaved in memory without causing jittery artifacts.

DDCAPS2_CERTIFIED

Indicates that display hardware is certified.

DDCAPS2_COLORCONTROLPRIMARY

The primary surface contains color controls (gamma, etc.)

DDCAPS2_COLORCONTROLOVERLAY

The overlay surface contains color controls (brightness, sharpness, etc.)

DDCAPS2_NO2DDURING3DSCENE

Indicates that 2-D operations such as **IDirectDrawSurface3::Blt** and **IDirectDrawSurface3::Lock** cannot be performed on any surfaces that Direct3D® is using between calls to the **IDirect3DDevice2::BeginScene** and **IDirect3DDevice2::EndScene** methods.

DDCAPS2_NONLOCALVIDMEM

Indicates that the display driver supports surfaces in non-local video memory.

DDCAPS2_NONLOCALVIDMEMCAPS

Indicates that blit capabilities for non-local video memory surfaces differ from local video memory surfaces. If this flag is present, the DDCAPS2_NONLOCALVIDMEM flag will also be present.

DDCAPS2_NOPAGELOCKREQUIRED

DMA blit operations are supported on system memory surfaces that are not page locked.

DDCAPS2_VIDEOPORT

Indicates that display hardware supports live video.

DDCAPS2_WIDESURFACES

Indicates that the display surfaces supports surfaces wider than the primary surface.

dwCKeyCaps

Color-key capabilities.

DDCKEYCAPS_DESTBLT

Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACE

Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for RGB colors.

DDCKEYCAPS_DESTBLTCLRSPACEYUV

Supports transparent blitting with a color space that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTBLTYUV

Supports transparent blitting with a color key that identifies the replaceable bits of the destination surface for YUV colors.

DDCKEYCAPS_DESTOVERLAY

Supports overlaying with color keying of the replaceable bits of the destination surface being overlaid for RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACE

Supports a color space as the color key for the destination of RGB colors.

DDCKEYCAPS_DESTOVERLAYCLRSPACEYUV

Supports a color space as the color key for the destination of YUV colors.

DDCKEYCAPS_DESTOVERLAYONEACTIVE

Supports only one active destination color key value for visible overlay surfaces .

DDCKEYCAPS_DESTOVERLAYYUV

Supports overlaying using color keying of the replaceable bits of the destination surface being overlaid for YUV colors.

DDCKEYCAPS_NOCOSTOVERLAY

Indicates there are no bandwidth trade-offs for using the color key with an overlay.

DDCKEYCAPS_SRCBLT

Supports transparent blitting using the color key for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACE

Supports transparent blitting using a color space for the source with this surface for RGB colors.

DDCKEYCAPS_SRCBLTCLRSPACEYUV

Supports transparent blitting using a color space for the source with this surface for YUV colors.

DDCKEYCAPS_SRCBLTYUV

Supports transparent blitting using the color key for the source with this surface for YUV colors.

DDCKEYCAPS_SRCOVERLAY

Supports overlaying using the color key for the source with this

overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACE

Supports overlaying using a color space as the source color key for the overlay surface for RGB colors.

DDCKEYCAPS_SRCOVERLAYCLRSPACEYUV

Supports overlaying using a color space as the source color key for the overlay surface for YUV colors.

DDCKEYCAPS_SRCOVERLAYONEACTIVE

Supports only one active source color key value for visible overlay surfaces.

DDCKEYCAPS_SRCOVERLAYYUV

Supports overlaying using the color key for the source with this overlay surface for YUV colors.

dwFXCaps

Driver-specific stretching and effects capabilities.

DDFXCAPS_BLTARITHSTRETCHY

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically).

DDFXCAPS_BLTARITHSTRETCHYN

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during a blit operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_BLTMIRRORLEFTRIGHT

Supports mirroring left to right in a blit operation.

DDFXCAPS_BLTMIRRORUPDOWN

Supports mirroring top to bottom in a blit operation.

DDFXCAPS_BLTROTATION

Supports arbitrary rotation in a blit operation.

DDFXCAPS_BLTROTATION90

Supports 90-degree rotations in a blit operation.

DDFXCAPS_BLTSHRINKX

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

DDFXCAPS_BLTSHRINKXN

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

DDFXCAPS_BLTSHRINKY

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

DDFXCAPS_BLTSHRINKYN

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

DDFXCAPS_BLTSTRETCHX

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

DDFXCAPS_BLTSTRETCHXN

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

DDFXCAPS_BLTSTRETCHY

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

DDFXCAPS_BLTSTRETCHYN

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

DDFXCAPS_OVERLAYARITHSTRETCHY

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during an overlay operation. Occurs along the y-axis (vertically).

DDFXCAPS_OVERLAYARITHSTRETCHYN

Uses arithmetic operations, rather than pixel-doubling techniques, to stretch and shrink surfaces during an overlay operation. Occurs along the y-axis (vertically), and works only for integer stretching (×1, ×2, and so on).

DDFXCAPS_OVERLAYMIRRORLEFTRIGHT

Supports mirroring of overlays around the vertical axis.

DDFXCAPS_OVERLAYMIRRORUPDOWN

Supports mirroring of overlays across the horizontal axis.

DDFXCAPS_OVERLAYSHRINKX

Supports arbitrary shrinking of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKXN

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

DDFXCAPS_OVERLAYSHRINKY

Supports arbitrary shrinking of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSHRINKYN

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

DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that shrinking is available.

DDFXCAPS_OVERLAYSTRETCHX

Supports arbitrary stretching of a surface along the x-axis (horizontally). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHXN

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

DDFXCAPS_OVERLAYSTRETCHY

Supports arbitrary stretching of a surface along the y-axis (vertically). This flag is valid only for DDSCAPS_OVERLAY surfaces. This flag indicates only the capabilities of a surface; it does not indicate that stretching is available.

DDFXCAPS_OVERLAYSTRETCHYN

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

dwFXAlphaCaps

Driver-specific alpha capabilities.

DDFXALPHACAPS_BLTALPHAEDGEBLEND

Supports alpha blending around the edge of a source color-keyed surface. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELS

Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHAPIXELSNEG

Supports alpha information in pixel format. The bit depth of alpha information in the pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA is set. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACES

Supports alpha-only surfaces. The bit depth of an alpha-only

surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for blit operations.

DDFXALPHACAPS_BLTALPHASURFACESNEG

Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be set only if DDCAPS_ALPHA has been set. Used for blit operations.

DDFXALPHACAPS_OVERLAYALPHAEDGEBLEND

Supports alpha blending around the edge of a source color-keyed surface. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELS

Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHAPIXELSNEG

Supports alpha information in pixel format. The bit depth of alpha information in pixel format can be 1, 2, 4, or 8. The alpha value becomes more transparent as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACES

Supports alpha-only surfaces. The bit depth of an alpha-only surface can be 1, 2, 4, or 8. The alpha value becomes more opaque as the alpha value increases. Regardless of the depth of the alpha information, 0 is always the fully transparent value. Used for overlays.

DDFXALPHACAPS_OVERLAYALPHASURFACESNEG

Indicates that the alpha channel becomes more transparent as the alpha value increases. The depth of the alpha channel data can be 1, 2, 4, or 8. Regardless of the depth of the alpha information, 0 is always the fully opaque value. This flag can be used only if DDCAPS_ALPHA has been set. Used for overlays.

dwPalCaps

Palette capabilities.

DDPCAPS_1BIT

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

DDPCAPS_2BIT

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

DDPCAPS_4BIT

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

color table.

DDPCAPS_8BIT

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

DDPCAPS_8BITENTRIES

Specifies an index to an 8-bit color index. This field is valid only when used with the DDPCAPS_1BIT, DDPCAPS_2BIT, or DDPCAPS_4BIT capability and when the target surface is in 8 bits per pixel (bpp). Each color entry is 1 byte long and is an index to an 8-bpp palette on the destination surface.

DDPCAPS_ALLOW256

Indicates that this palette can have all 256 entries defined.

DDPCAPS_PRIMARYSURFACE

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

DDPCAPS_PRIMARYSURFACELEFT

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

DDPCAPS_VSYNC

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

dwSVCaps

Stereo vision capabilities.

DDSVCAPS_ENIGMA

Indicates that the stereo view is accomplished using Enigma encoding.

DDSVCAPS_FLICKER

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

DDSVCAPS_REDBLUE

Indicates that the stereo view is accomplished when the viewer looks at the image through red and blue filters placed over the left and right eyes. All images must adapt their color spaces for this process.

DDSVCAPS_SPLIT

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

dwAlphaBltConstBitDepths

DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaBltPixelBitDepths

DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaBltSurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayConstBitDepths
DDBD_2, DDBD_4, or DDBD_8. (Indicates 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlayPixelBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwAlphaOverlaySurfaceBitDepths
DDBD_1, DDBD_2, DDBD_4, or DDBD_8. (Indicates 1-, 2-, 4-, or 8-bits per pixel.)

dwZBufferBitDepths
DDBD_8, DDBD_16, or DDBD_24. (Indicates 8-, 16-, 24-bits per pixel.) 32-bit z-buffers are not supported.

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 for an overlay surface, in pixels.

dwAlignSizeSrc
Source rectangle size alignment for an overlay surface, in pixels. Overlay source rectangles must have a pixel width that is a multiple of this value.

dwAlignBoundaryDest
Destination rectangle alignment for an overlay surface, in pixels.

dwAlignSizeDest
Destination rectangle size alignment for an overlay surface, in pixels. Overlay destination rectangles must have a pixel width that is a multiple of this value.

dwAlignStrideAlign
Stride alignment.

dwRops[DD_ROP_SPACE]
Raster operations supported.

ddsCaps
DDSCAPS structure with general capabilities.

dwMinOverlayStretch and **dwMaxOverlayStretch**
Minimum and maximum overlay stretch factors multiplied by 1000. For example, 1.3 = 1300.

dwMinLiveVideoStretch and dwMaxLiveVideoStretch
These members are obsolete; do not use.

dwMinHwCodecStretch and **dwMaxHwCodecStretch**
These members are obsolete; do not use.

dwReserved1, **dwReserved2**, and **dwReserved3**
Reserved for future use.

dwSVBCaps
Driver-specific capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwSVBCKeyCaps
Driver color-key capabilities for system-memory-to-display-memory blits. Valid flags are identical to

the blit-related flags used with for the **dwCKeyCaps** member.

dwSVBFXCaps
Driver FX capabilities for system-memory-to-display-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwSVBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-display-memory blits.

dwVSBCaps
Driver-specific capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwVSBCKeyCaps
Driver color-key capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwVSBFXCaps
Driver FX capabilities for display-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwVSBRops[DD_ROP_SPACE]
Raster operations supported for display-memory-to-system-memory blits.

dwSSBCaps
Driver-specific capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwSSBCKeyCaps
Driver color-key capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwSSBCFXCaps
Driver FX capabilities for system-memory-to-system-memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwSSBRops[DD_ROP_SPACE]
Raster operations supported for system-memory-to-system-memory blits.

dwMaxVideoPorts
Maximum number of live video ports.

dwCurrVideoPorts
Current number of live video ports.

dwSVBCaps2
More driver-specific capabilities for system-memory-to-video-memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps2** member.

dwNLVBCaps
Driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps** member.

dwNLVBCaps2
More driver-specific capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwCaps2** member.

dwNLVBCKeyCaps
Driver color-key capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with for the **dwCKeyCaps** member.

dwNLVBFXCaps
Driver FX capabilities for nonlocal-to-local video memory blits. Valid flags are identical to the blit-related flags used with the **dwFXCaps** member.

dwNLVBRops[DD_ROP_SPACE]
Raster operations supported for nonlocal-to-local video memory blits.

dwReserved4, dwReserved5, and dwReserved6

Reserved for future use.

DDCOLORCONTROL

The **DDCOLORCONTROL** structure defines the color controls associated with a DirectDrawVideoPortObject, an overlay surface, or a primary surface.

```
typedef struct _DDCOLORCONTROL {
    DWORD   dwSize;
    DWORD   dwFlags;
    LONG    lBrightness;
    LONG    lContrast;
    LONG    lHue;
    LONG    lSaturation;
    LONG    lSharpness;
    LONG    lGamma;
    LONG    lColorEnable;
    DWORD   dwReserved1;
} DDCOLORCONTROL, FAR *LPDDCOLORCONTROL;
```

Members

dwSize

The the size of the structure, in bytes. This member must be initialized before use.

dwFlags

Flags specifying which structure members contain valid data . When the structure is returned by the **IDirectDrawColorControl::GetColorControls** method, it also indicates which options are supported by the device.

DDCOLOR_BRIGHTNESS	The **iBrightness** member contains valid data.
DDCOLOR_CONTRAST	The **iContrast** member contains valid data.
DDCOLOR_COLORENABLE	The **lColorEnable** member contains valid data.
DDCOLOR_GAMMA	The **iGamma** member contains valid data.
DDCOLOR_HUE	The **iHue** member contains valid data.
DDCOLOR_SATURATION	The **iSaturation** member contains valid data.
DDCOLOR_SHARPNESS	The **iSharpness** member contains valid data.

lBrightness

Luminance intensity (Black Level) in IRE units*100. Range is 0 to 10,000. The default is 750 (7.5 IRE)

lContrast

Relative difference between higher and lower intensity luminance values in IRE units*100. The valid range is 0 to 20,000. The default value is 10,000 (100 IRE). Higher values of contrast cause darker luminance values to tend towards black, and cause lighter luminance values to tend towards white. Lower values of contrast cause all luminance values to move towards the middle luminance values.

lHue

Phase relationship of the chrominance components. Hue is specified in degrees and the valid range is -180 to 180. The default is 0.

lSaturation

Color intensity in IRE units*100. The valid range is 0 to 20,000. The default value is 10,000 (100

IRE).

lSharpness
Sharpness in arbitrary units. The valid range is 0 to 10. The default value is 5.

lGamma
Controls the amount of gamma correction applied to the luminance values. The valid range is 1 to 500 gamma units, with a default of 1.

lColorEnable
Flag indicating whether color is used. If this member is zero, color is not used; if it is 1, then color is used. The default value is 1.

dwReserved1
This member is reserved.

DDCOLORKEY

The **DDCOLORKEY** structure describes a source color key, destination color key, or color space. A color key is specified if the low and high range values are the same. This structure is used with the **IDirectDrawSurface3::GetColorKey** and **IDirectDrawSurface3::SetColorKey** methods.

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

Members

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

The **DDOVERLAYFX** structure passes override information to the **IDirectDrawSurface3::UpdateOverlay** method.

```
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;
```

Members

dwSize
Size of the structure, in bytes. This members must be initialized before the structure is used.

dwAlphaEdgeBlendBitDepth
Bit depth used to specify the constant for an alpha edge blend.

dwAlphaEdgeBlend
Constant to use as the alpha for an edge blend.

dwReserved
Reserved for future use.

dwAlphaDestConstBitDepth
Bit depth used to specify the alpha constant for a destination.

dwAlphaDestConst
Constant to use as the alpha channel for a destination.

lpDDSAlphaDest
Address of a surface to use as the alpha channel for a destination.

dwAlphaSrcConstBitDepth
Bit depth used to specify the alpha constant for a source.

dwAlphaSrcConst
Constant to use as the alpha channel for a source.

lpDDSAlphaSrc
Address of a surface to use as the alpha channel for a source.

dckDestColorkey
Destination color key override.

dckSrcColorkey
Source color key override.

dwDDFX
Overlay FX flags.

DDOVERFX_ARITHSTRETCHY
If stretching, use arithmetic stretching along the y-axis for this overlay.

DDOVERFX_MIRRORLEFTRIGHT
Mirror the overlay around the vertical axis.

DDOVERFX_MIRRORUPDOWN
Mirror the overlay around the horizontal axis.

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

DDPIXELFORMAT

The **DDPIXELFORMAT** structure describes the pixel format of a DirectDrawSurface object for the **IDirectDrawSurface3::GetPixelFormat** method.

```
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;
        DWORD dwRGBZBitMask;
        DWORD dwYUVZBitMask;
    };
} DDPIXELFORMAT, FAR* LPDDPIXELFORMAT;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Optional control flags.

DDPF_ALPHA

The pixel format describes an alpha-only surface.

DDPF_ALPHAPIXELS

The surface has alpha channel information in the pixel format.

DDPF_COMPRESSED

The surface will accept pixel data in the specified format and compress it during the write operation.

DDPF_FOURCC

The **dwFourCC** member is valid and contains a FOURCC code describing a non-RGB pixel format..

DDPF_PALETTEINDEXED1

DDPF_PALETTEINDEXED2

DDPF_PALETTEINDEXED4

DDPF_PALETTEINDEXED8

The surface is 1-, 2-, 4-, or 8-bit color indexed.

DDPF_PALETTEINDEXEDTO8

The surface is 1-, 2-, or 4-bit color indexed to an 8-bit palette.

DDPF_RGB

The RGB data in the pixel format structure is valid.

DDPF_RGBTOYUV

The surface will accept RGB data and translate it during the write operation to YUV data. The format of the data to be written will be contained in the pixel format structure. The DDPF_RGB flag will be set.

DDPF_YUV

The YUV data in the pixel format structure is valid.

DDPF_ZBUFFER

The pixel format describes a z-buffer-only surface.

DDPF_ZPIXELS

The surface is in RGBZ format.

dwFourCC

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

dwRGBBitCount

RGB bits per pixel (4, 8, 16, 24, or 32).

dwYUVBitCount

YUV bits per pixel (4, 8, 16, 24, or 32)

dwZBufferBitDepth

Z-buffer bit depth (8, 16, or 24). 32-bit z-buffers are not supported.

dwAlphaBitDepth

Alpha channel bit depth (1, 2, 4, or 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 and **dwYUVAlphaBitMask**

Masks for alpha channel.

dwRGBZBitMask and **dwYUVZBitMask**
Masks for z channel.

DDSCAPS

The **DDSCAPS** structure defines the capabilities of a DirectDrawSurface object. This structure is part of the **DDCAPS** structure that is used to describe the capabilities of the DirectDraw object.

```
typedef struct _DDSCAPS{
        DWORD dwCaps;
} DDSCAPS,FAR* LPDDSCAPS;
```

Members

dwCaps

Capabilities of the surface. One or more of the following flags:

DDSCAPS_3D

Unsupported. Use the DDSCAPS_3DDEVICE instead.

DDSCAPS_3DDEVICE

Indicates that this surface can be used for 3-D rendering. Applications can use this flag to ensure that a device that can only render to a certain heap has off-screen surfaces allocated from the correct heap. If this flag is set for a heap, the surface is not allocated from that heap.

DDSCAPS_ALLOCONLOAD

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

DDSCAPS_ALPHA

Indicates that this surface contains alpha-only information.

DDSCAPS_BACKBUFFER

Indicates that this surface is the back buffer of a surface flipping structure. Typically, this capability is set by the **IDirectDraw2::CreateSurface** method when the DDSCAPS_FLIP flag is used. Only the surface that immediately precedes the DDSCAPS_FRONTBUFFER surface has this capability set. The other surfaces are identified as back buffers by the presence of the DDSCAPS_FLIP flag, their attachment order, and the absence of the DDSCAPS_FRONTBUFFER and DDSCAPS_BACKBUFFER capabilities. If this capability is sent to the **IDirectDraw2::CreateSurface** method, a stand-alone back buffer is being created. After this method is called, this surface could be attached to a front buffer, another back buffer, or both to form a flipping surface structure. For more information, see **IDirectDrawSurface3::AddAttachedSurface**. DirectDraw supports an arbitrary number of surfaces in a flipping structure.

DDSCAPS_COMPLEX

Indicates that a complex surface is being described. A complex surface results in the creation of more than one surface. The additional surfaces are attached to the root surface. The complex structure can be destroyed only by destroying the root.

DDSCAPS_FLIP

Indicates that this surface is a part of a surface flipping structure. When this capability is passed to the **IDirectDraw2::CreateSurface** method, a front buffer and one or

more back buffers are created. DirectDraw sets the DDSCAPS_FRONTBUFFER bit on the front-buffer surface and the DDSCAPS_BACKBUFFER bit on the surface adjacent to the front-buffer surface. The **dwBackBufferCount** member of the **DDSURFACEDESC** structure must be set to at least 1 in order for the method call to succeed. The DDSCAPS_COMPLEX capability must always be set when creating multiple surfaces by using the **IDirectDraw2::CreateSurface** method.

DDSCAPS_FRONTBUFFER

Indicates that this surface is the front buffer of a surface flipping structure. This flag is typically set by the **IDirectDraw2::CreateSurface** method when the DDSCAPS_FLIP capability is set. If this capability is sent to the **IDirectDraw2::CreateSurface** method, a stand-alone front buffer is created. This surface will not have the DDSCAPS_FLIP capability. It can be attached to other back buffers to form a flipping structure by using **IDirectDrawSurface3::AddAttachedSurface**.

DDSCAPS_HWCODEC

Indicates that this surface should be able to have a stream decompressed to it by the hardware.

DDSCAPS_LIVEVIDEO

Indicates that this surface should be able to receive live video.

DDSCAPS_LOCALVIDMEM

Indicates that this surface exists in true, local video memory rather than non-local video memory. If this flag is specified then DDSCAPS_VIDEOMEMORY must be specified as well. This flag cannot be used with the DDSCAPS_NONLOCALVIDMEM flag.

DDSCAPS_MIPMAP

Indicates that this surface is one level of a mipmap. This surface will be attached to other DDSCAPS_MIPMAP surfaces to form the mipmap. This can be done explicitly by creating a number of surfaces and attaching them by using the **IDirectDrawSurface3::AddAttachedSurface** method, or implicitly by the **IDirectDraw2::CreateSurface** method. If this capability is set, DDSCAPS_TEXTURE must also be set.

DDSCAPS_MODEX

Indicates that this surface is a 320×200 or 320×240 Mode X surface.

DDSCAPS_NONLOCALVIDMEM

Indicates that this surface exists in nonlocal video memory rather than true, local video memory. If this flag is specified, then DDSCAPS_VIDEOMEMORY flag must be specified as well. This cannot be used with the DDSCAPS_LOCALVIDMEM flag.

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_OPTIMIZED

Not currently implemented.

DDSCAPS_OVERLAY

Indicates that this surface is an overlay. It may or may not be directly visible depending on whether it is currently being overlaid onto the primary surface. DDSCAPS_VISIBLE can be used to determine if it is being overlaid at the moment.

DDSCAPS_OWNDC

Indicates that this surface will have a device context (DC) association for a long period.

DDSCAPS_PALETTE

Indicates that this device driver allows unique DirectDrawPalette objects to be created and attached to this surface.

DDSCAPS_PRIMARYSURFACE

Indicates the surface is the primary surface. It represents what is visible to the user at the moment.

DDSCAPS_PRIMARYSURFACELEFT

Indicates that this surface is the primary surface for the left eye. It represents what is visible to the user's left eye at the moment. When this surface is created, the surface with the DDSCAPS_PRIMARYSURFACE capability represents what is seen by the user's right eye.

DDSCAPS_STANDARDVGAMODE

Indicates that this surface is a standard VGA mode surface, and not a Mode X surface. This flag cannot be used in combination with the DDSCAPS_MODEX flag.

DDSCAPS_SYSTEMMEMORY

Indicates that this surface memory was allocated in system memory.

DDSCAPS_TEXTURE

Indicates that this surface can be used as a 3-D texture. It does not indicate whether the surface is being used for that purpose.

DDSCAPS_VIDEOMEMORY

Indicates that this surface exists in display memory.

DDSCAPS_VIDEOPORT

Indicates that this surface can receive data from a video port.

DDSCAPS_VISIBLE

Indicates that changes made to this surface are immediately visible. It is always set for the primary surface, as well as for overlays while they are being overlaid and texture maps while they are being textured.

DDSCAPS_WRITEONLY

Indicates that only write access is permitted to the surface. Read access from the surface may generate a general protection (GP) fault, but the read results from this surface will not be meaningful.

DDSCAPS_ZBUFFER

Indicates that this surface is the z-buffer. The z-buffer contains information that cannot be displayed. Instead, it contains bit-depth information that is used to determine which pixels are

visible and which are obscured.

DDSURFACEDESC

The **DDSURFACEDESC** structure contains a description of the surface to be created. This structure is passed to the **IDirectDraw2::CreateSurface** method. The relevant members differ for each potential type of surface.

```
typedef struct _DDSURFACEDESC {
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwHeight;
    DWORD           dwWidth;
    union
    {
        LONG        lPitch;
        DWORD       dwLinearSize;
    };
    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;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Optional control flags. One or more of the following flags:

DDSD_ALL

Indicates that all input members are valid.

DDSD_ALPHABITDEPTH

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

DDSD_BACKBUFFERCOUNT

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

DDSD_CAPS

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

DDSD_CKDESTBLT

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

DDSD_CKDESTOVERLAY

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

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

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

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

DDSD_LINEARSIZE
: Not used.

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

DDSD_MIPMAPCOUNT
: Indicates that the **dwMipMapCount** member is valid.

DDSD_PITCH
: Indicates that the **lPitch** member is valid.

DDSD_PIXELFORMAT
: Indicates that the **ddpfPixelFormat** member is valid.

DDSD_REFRESHRATE
: Indicates that the **dwRefreshRate** member is valid.

DDSD_WIDTH
: Indicates that the **dwWidth** member is valid.

DDSD_ZBUFFERBITDEPTH
: Indicates that the **dwZBufferBitDepth** member is valid.

dwHeight and **dwWidth**
: Dimensions of the surface to be created, in pixels.

lPitch
: Distance, in bytes, to the start of next line. When used with the **IDirectDrawSurface3::GetSurfaceDesc** method, this is a return value. When used with the **IDirectDrawSurface3::SetSurfaceDesc** method, this is an input value that must be a DWORD multiple.

dwLinearSize
: Not currently used.

dwBackBufferCount
: Number of back buffers.

dwMipMapCount
: Number of mipmap levels.

dwZBufferBitDepth
: Depth of z-buffer. 32-bit z-buffers are not supported.

dwRefreshRate
: Refresh rate (used when the display mode is described). The value of 0 indicates an adapter fault.

dwAlphaBitDepth
: Depth of alpha buffer.

dwReserved
: Reserved.

lpSurface
: Address of the associated surface memory. When calling **IDirectDrawSurface3::Lock**, this member is a valid pointer to surface memory. When calling **IDirectDrawSurface3::SetSurfaceDesc**, this

member is a pointer to system memory that the caller explicitly allocates for the DirectDrawSurface object.

ddckCKDestOverlay
DDCOLORKEY structure that describes the destination color key to be used for an overlay surface.

ddckCKDestBlt
DDCOLORKEY structure that describes the destination color key for blit operations.

ddckCKSrcOverlay
DDCOLORKEY structure that describes the source color key to be used for an overlay surface.

ddckCKSrcBlt
DDCOLORKEY structure that describes the source color key for blit operations.

ddpfPixelFormat
DDPIXELFORMAT structure that describes the surface's pixel format.

ddsCaps
DDSCAPS structure containing the surface's capabilities.

DDVIDEOPORTBANDWIDTH

The **DDVIDEOPORTBANDWIDTH** structure describes the bandwidth characteristics of an overlay surface when used with a particular video port and pixel format configuration. This structure is used with the **IDirectDrawVideoPort::GetBandwidthInfo** method.

```
typedef struct _DDVIDEOPORTBANDWIDTH {
    DWORD dwSize;          // Size of the DDVIDEOPORTBANDWIDTH structure
    DWORD dwCaps;          // Caps flags
    DWORD dwOverlay;       // Zoom factor at which overlay is supported
    DWORD dwColorkey;      // Zoom factor at which overlay w/ colorkey is
supported
    DWORD dwYInterpolate;  // Zoom factor at which overlay w/ Y
interpolation is supported
    DWORD dwYInterpAndColorkey;  // Zoom factor at which overlay w/ Y
interpolation and colorkeying is supported
    DWORD dwReserved1;     // Reserved for future use - set to zero
    DWORD dwReserved2;     // Reserved for future use - set to zero
} DDVIDEOPORTBANDWIDTH,*LPDDVIDEOPORTBANDWIDTH;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

dwCaps
Flag values specifying device dependency. This member can be one of the following values.

DDVPBCAPS_DESTINATION	This device's capabilities are described in terms of the overlay's minimum stretch factor. Bandwidth information provided for this device refers to the destination overlay size.
DDVPBCAPS_SOURCE	This device's capabilities are described in terms of the required source overlay size. Bandwidth information provided for this device refers to the source overlay size.

dwOverlay
Stretch factor or overlay source size at which an overlay is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwColorkey
Stretch factor or overlay source size at which an overlay with color keying is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwYInterpolate
Stretch factor or overlay source size at which an overlay with Y-axis interpolation is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwYInterpolateAndColorkey
Stretch factor or overlay source size at which an overlay with Y-axis interpolation and color keying is supported multiplied by 1000. For example 1.3 = 1300, or .75 = 750.

dwReserved1 and **dwReserved2**
Reserved; set to zero.

Remarks

When DDVPBCAPS_DESTINATION is specified, the stretch factors described in the other members describe the minimum stretch factor required to display an overlay with the dimensions given when

calling the **GetBandwidthInfo** method. Stretch factor values under 1000 mean that the video port is capable of shrinking an overlay when displayed, and values over 1000 mean that the overlay must be stretched larger than their source to be displayed.

When DDVPBCAPS_SOURCE is specified, the stretch factors described in the other members describe how much you must shrink the overlay source in order for it to be displayed. In this case, the best possible value is 1000, meaning that no shrinking is required. Smaller values tell you that the source rectangle you specified when calling **GetBandwidthInfo** were too large and must be smaller. For example, if the stretch factor is 750 and you specified 320 pixels for the *dwWidth* parameter, then you will not be able to display the overlay at that size. To successfully display the overlay, you must use a source rectangle 240 pixels wide to successfully display the overlay.

DDVIDEOPORTCAPS

The **DDVIDEOPORTCAPS** structure describes the capabilities and alignment restrictions of a video port. This structure is used with the **IDDVideoPortContainer::EnumVideoPorts** method.

```
typedef struct _DDVIDEOPORTCAPS {
    DWORD dwSize;           // Size of the DDVIDEOPORTCAPS structure
    DWORD dwFlags;          // Indicates which fields contain data
    DWORD dwMaxWidth;       // Max width of the video port field
    DWORD dwMaxVBIWidth;    // Max width of the VBI data
    DWORD dwMaxHeight;      // Max height of the video port field
    DWORD dwVideoPortID;    // Video port ID (0 - (dwMaxVideoPorts -1))
    DWORD dwCaps;           // Video port capabilities
    DWORD dwFX;             // More video port capabilities
    DWORD dwNumAutoFlipSurfaces;          // Number of autoflippable
surfaces
    DWORD dwAlignVideoPortBoundary;       // Byte restriction of placement
within the surface
    DWORD dwAlignVideoPortPrescaleWidth;  // Byte restriction of width after
prescaling
    DWORD dwAlignVideoPortCropBoundary;   // Byte restriction of left
cropping
    DWORD dwAlignVideoPortCropWidth;      // Byte restriction of cropping
width
    DWORD dwPreshrinkXStep;           // Width can be shrunk in steps of 1/x
    DWORD dwPreshrinkYStep;           // Height can be shrunk in steps of
1/x
    DWORD dwNumVBIAutoFlipSurfaces;   // Number of VBI autoflippable
surfaces
    DWORD dwReserved1;                // Reserved for future use
    DWORD dwReserved2;                // Reserved for future use
} DDVIDEOPORTCAPS, *LPDDVIDEOPORTCAPS;
```

Members

dwSize

Size of the structure, in bytes. This must be initialized before use.

dwFlags

Flag values indicating the fields that contain valid data. The following flags are defined.

DDVD_WIDTH	The **dwMaxWidth** member is valid.
DDVPD_HEIGHT	The **dwMaxHeight** member is valid.
DDVPD_ID	The **dwVideoPortID** member is valid.
DDVPD_CAPS	The **dwCaps** member is valid.
DDVPD_FX	The **dwFX** member is valid.
DDVPD_AUTOFLIP	The **dwNumAutoFlipSurfaces** member is valid.
DDVPD_ALIGN	The **dwAlignVideoPortBoundary**, **dwAlignVideoPortPrescaleWidth**,**dwAlignVideo PortCropBoundary,** and **dwAlignVideoPortCropWidth** are valid.

dwMaxWidth

Maximum width of the video port field.

dwMaxVBIWidth

Maximum width of the VBI data.

dwMaxHeight

Maximum height of the video port field.

dwVideoPortID

Zero based index identifying the video port.

dwCaps

Video port capabilities.

DDVPCAPS_AUTOFLIP	Flip can be performed automatically to avoid tearing when a VREF occurs. If the data is being interleaved in memory, it will flip on every other VREF.
DDVPCAPS_INTERLACED	Supports interlaced video.
DDVPCAPS_NONINTERLACED	Supports non-interlaced video.
DDVPCAPS_READBACKFIELD	Supports the **IDirectDrawVideoPort::GetFieldPolarity** method.
DDVPCAPS_READBACKLINE	Supports the **IDirectDrawVideoPort::GetVideoLine** method.
DDVPCAPS_SHAREABLE	Supports two genlocked video streams that share the video port, where one stream uses the even fields and the other uses the odd fields. Separate parameters (including address, scaling, cropping, etc.) are maintained for both fields.
DDVPCAPS_SKIPEVENFIELDS	Even fields of video can be automatically discarded.
DDVPCAPS_SKIPODDFIELDS	Odd fields of video can be automatically discarded.
DDVPCAPS_SYNCMASTER	Can drive the graphics sync (refresh rate) based on the video port sync.
DDVPCAPS_SYSTEMMEMORY	Capable of writing to surfaces created in system memory.
DDVPCAPS_VBISURFACE	Data within the VBI can be written to a different surface.
DDVPCAPS_COLORCONTROL	Can perform color control operations on incoming data before writing to the frame buffer.
DDVPCAPS_OVERSAMPLEDVBI	Can accept VBI data in a different format or width than the regular video data.

dwFX

Additional video port capabilities.

DDVPFX_CROPTOPDATA	Limited cropping is available to crop VBI data.
DDVPFX_CROPX	Incoming data can be cropped in the x direction before it is written to the surface.
DDVPFX_CROPY	Incoming data can be cropped in the y direction before it is written to the surface.
DDVPFX_INTERLEAVE	Supports interleaving interlaced fields in memory.
DDVPFX_MIRRORLEFTRIGHT	Supports mirroring left to right as the video data is written into the frame buffer.
DDVPFX_MIRRORUPDOWN	Supports mirroring top to bottom as the video data is written into the frame buffer.

DDVPFX_PRESHRINKX	Data can be arbitrarily shrunk in the x direction before it is written to the surface.
DDVPFX_PRESHRINKY	Data can be arbitrarily shrunk in the y direction before it is written to the surface.
DDVPFX_PRESHRINKXB	Data can be binary shrunk (1/2, 1/4, 1/8, etc.) in the x direction before it is written to the surface.
DDVPFX_PRESHRINKYB	Data can be binary shrunk (1/2, 1/4, 1/8, etc.) in the y direction before it is written to the surface.
DDVPCAPS_PRESHRINKXS	Data can be shrunk in the x direction by increments of *1/x*, where *x* is specified in the **dwShrinkXStep** member.
DDVPCAPS_PRESHRINKYS	Data can be shrunk in the y direction by increments of *1/y*, where *y* is specified in the **dwShrinkYStep**
DDVPFX_PRESTRETCHX	Data can be arbitrarily stretched in the x direction before it is written to the surface.
DDVPFX_PRESTRETCHY	Data can be arbitrarily stretched in the y direction before it is written to the surface.
DDVPFX_PRESTRETCHXN	Data can be integer stretched in the x direction before it is written to the surface. (1x, 2x, 3x, etc.)
DDVPFX_PRESTRETCHYN	Data can be integer stretched in the y direction before it is written to the surface. (1x, 2x, 3x, etc.)
DDVPFX_VBICONVERT	Data within the VBI can be converted independently of the remaining video data.
DDVPFX_VBINOSCALE	Scaling can be disabled for data within the VBI.
DDVPFX_IGNOREVBIXCROP	The video port can ignore the left and right cropping coordinates when cropping oversampled VBI data.

dwNumAutoFlipSurfaces
Number of autoflippable surfaces supported by the video port.

dwAlignVideoPortBoundary
Byte restriction of placement within the surface.

dwAlignVideoPortPrescaleWidth
Byte restriction of width after prescaling.

dwAlignVideoPortCropBoundary
Byte restriction of left cropping.

dwAlignVideoPortCropWidth
Byte restriction of cropping width.

dwPreshrinkXStep
Width can be shrunk in the x direction in steps of 1/**dwPreshrinkXStep**.

dwPreshrinkYStep
Height can be shrunk in the y direction in steps of *1/***dwPreshrinkYStep**.

dwNumVBIAutoFlipSurfaces
Number of autoflipping surfaces capable of receiving data transmitted during the vertical blanking interval (VBI) independent from the remainder of the video stream. When constructing the autoflip chain, the number of VBI surfaces must equal the number of surfaces receiving the remainder of the video data.

dwReserved1 and **dwReserved2**
Reserved; set to zero.

DDVIDEOPORTCONNECT

The DDVIDEOPORTCONNECT structure describes a video port connection. This structure is used with the **IDDVideoPortContainer::GetVideoPortConnectInfo** method.

```
typedef struct _DDVIDEOPORTCONNECT{
    DWORD dwSize;           // Size of the DDVIDEOPORTCONNECT structure
    DWORD dwPortWidth;      // Width of the video port
    GUID  guidTypeID;       // Description of video port connection
    DWORD dwFlags;          // Connection flags
    DWORD dwReserved1;      // Reserved, set to zero.
} DDVIDEOPORTCONNECT,*LPDDVIDEOPORTCONNECT;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before use.

dwPortWidth

Width of the video port. This value represents the number of physical pins on the video port, not the width of a surface in memory. This member must always be set, even when the value in the **guidTypeID** member assumes a certain size.

guidTypeID

A GUID that describes the sync characteristics of the video port. The following port types are predefined:

DDVPTYPE_E_HREFH_VREFH	External syncs where HREF is active high and VREF is active high.
DDVPTYPE_E_HREFH_VREFL	External syncs where HREF is active high and VREF is active low.
DDVPTYPE_E_HREFL_VREFH	External syncs where HREF is active low and VREF is active high.
DDVPTYPE_E_HREFL_VREFL	External syncs where HREF is active low and VREF is active low.
DDVPTYPE_CCIR656	Sync information is embedded in the data stream according to the CCIR656 spec.
DDVPTYPE_BROOKTREE	Sync information is embedded in the data stream using the Brooktree definition.
DDVPTYPE_PHILIPS	Sync information is embedded in the data stream using the Philips definition.

dwFlags

Flags describing the capabilities of the video-port connection. This member can be set by DirectDraw when connection information is being retrieved or by the client when connection information is being set. This member can be a combination of the following flags.

DDVPCONNECT_DOUBLECLOCK	Indicates that the video port either supports double-clocking data or should double-clock data. This flag is only valid with an external sync.
DDVPCONNECT_VACT	Indicates that the video port either supports using an external VACT signal or should use the external VACT signal. This flag is only valid with an external sync.
DDVPCONNECT_INVERTPOLARITY	Indicates that the video port is capable of inverting the field polarities or is to invert field

	polarities. When a video port inverts field polarities, it treats even fields as odd fields and vice versa.
DDVPCONNECT_DISCARDSVREFDATA	The video port discards any data written during the VREF period, causing it to not be written to the frame buffer. This flag is read-only.
DDVPCONNECT_HALFLINE	The video port will write half lines into the frame buffer, sometimes causing the data to be displayed incorrectly. This flag is read-only.
DDVPCONNECT_INTERLACED	Indicates that the signal is interlaced. This flag is only used by the client when creating a video port object.
DDVPCONNECT_SHAREEVEN	The physical video port is shareable, and that this video port object will use the even fields. This flag is only used by the client when creating the video port object.
DDVPCONNECT_SHAREODD	The physical video port is shareable, and that this video port object will use the odd fields. This flag is only used by the client when creating the video port object.

dwReserved1
Reserved; set to zero.

Remarks

This structure is used independently and as a member of the **DDVIDEOPORTDESC** structure.

DDVIDEOPORTDESC

The **DDVIDEOPORTDESC** structure describes a video-port object to be created. This structure is used with the **IDDVideoPortContainer::CreateVideoPort** method.

```
typedef struct _DDVIDEOPORTDESC {
    DWORD dwSize;                 // Size of the DDVIDEOPORTDESC structure.
    DWORD dwFieldWidth;           // Width of the video port field.
    DWORD dwVBIWidth;             // Width of the VBI data.
    DWORD dwFieldHeight;          // Height of the video port field.
    DWORD dwMicrosecondsPerField; // Microseconds per video field.
    DWORD dwMaxPixelsPerSecond;   // Maximum pixel rate per second.
    DWORD dwVideoPortID;          // Video port ID (0 - (dwMaxVideoPorts
-1)).
    DWORD dwReserved1;            // Reserved for future use - set to zero.
    DDVIDEOPORTCONNECT VideoPortType;  // Description of video port
connection.
    DWORD dwReserved2;            // Reserved for future use - set to zero.
    DWORD dwReserved3;            // Reserved for future use - set to zero.
 } DDVIDEOPORTDESC, *LPDDVIDEOPORTDESC;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

dwFieldWidth
Width of incoming video stream, in pixels.

dwVBIWidth
Width of the VBI data in the incoming video stream, in pixels.

dwFieldHeight
Field height for fields in the incoming video stream, in scan lines.

dwMicrosecondsPerField
Time interval, in microseconds, between live video VREF periods. This number should be rounded up to the nearest microsecond.

dwVideoPortID
The zero-based ID of the physical video port to be used.

dwReserved1
Reserved; set to zero.

VideoPortType
A **DDVIDEOPORTCONNECT** structure describing the connection characteristics of the video port.

dwReserved2 and **dwReserved3**
Reserved; set to zero.

DDVIDEOPORTINFO

The DDVIDEOPORTINFO structure describes the transfer of video data to a surface. This structure is used with the **IDirectDrawVideoPort::StartVideo** method.

```
typedef struct _DDVIDEOPORTINFO{
    DWORD dwSize;            // Size of the structure.
    DWORD dwOriginX;         // Placement of the video data within the
surface.
    DWORD dwOriginY;         // Placement of the video data within the
surface.
    DWORD dwVPFlags;         // Video port options.
    RECT rCrop;              // Cropping rectangle (optional).
    DWORD dwPrescaleWidth;   // Pre-scaling/zooming in the X direction
(optional).
    DWORD dwPrescaleHeight;  // Pre-scaling/zooming in the Y direction
(optional).
    LPDDPIXELFORMAT lpddpfInputFormat;     // Video format written to the
video port.
    LPDDPIXELFORMAT lpddpfVBIInputFormat;  // Input format of the VBI data.
    LPDDPIXELFORMAT lpddpfVBIOutputFormat; // Output format of the data.
    DWORD dwVBIHeight;        // Lines of data within the vertical blanking
interval.
    DWORD dwReserved1;        // Reserved for future use - set to zero.
    DWORD dwReserved2;        // Reserved for future use - set to zero.
} DDVIDEOPORTINFO,*LPDDVIDEOPORTINFO;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized before use.

dwOriginX and **dwOriginY**

X and y coordinates for the origin of the video data in the surface.

dwVPFlags

Video port options.

DDVP_AUTOFLIP	Perform automatic flipping. For more information, see Auto-flipping.
DDVP_CONVERT	Perform conversion using the information in the **ddpfOutputFormat** member.
DDVP_CROP	Perform cropping using the rectangle specified by the **rCrop** member.
DDVP_INTERLEAVE	Interlaced fields should be interleaved in memory.
DDVP_MIRRORLEFTRIGHT	Mirror image data from left to right as it is written into the frame buffer.
DDVP_MIRRORUPDOWN	Mirror image data from top to bottom as it is written into the frame buffer.
DDVP_PRESCALE	Perform pre-scaling or pre-zooming based on the values in the **dwPrescaleHeight** and **dwPrescaleWidth** members.
DDVP_SKIPEVENFIELDS	Ignore input of even fields.
DDVP_SKIPODDFIELDS	Ignore input of odd fields.
DDVP_SYNCMASTER	Indicates that the video port VREF should drive the

	graphics VREF, locking the refresh rate to the video port.
DDVP_VBICONVERT	The **ddpfVBIOutputFormat** member contains data that should be used to convert VBI data.
DDVP_VBINOSCALE	VBI data should not be scaled.
DDVPCONNECT_OVERRIDEBOBWEAVE	Override automatic display method chosen by the driver, using only the display method set by the caller when creating the overlay surface.
DDVPFX_IGNOREVBIXCROP	Indicates that the video port should ignore left and right cropping coordinates when cropping oversampled VBI data.

rCrop
Cropping rectangle. This member is optional.

dwPrescaleWidth
Pre-scaling or zooming in the x direction. This member is optional.

dwPrescaleHeight
Pre-scaling or zooming in the y direction. This member is optional.

ddpfInputFormat
A **DDPIXELFORMAT** structure describing the pixel format to be written to the video port. This will often be identical to the surface's pixel format, but can differ if the video port is to perform conversion.

ddpfVBIInputFormat and **ddpfVBIOutputFormat**
DDPIXELFORMAT structures describing the input and output pixel formats of the data within the vertical blanking interval.

dwVBIHeight
The amount of data within the vertical blanking interval, in scan lines.

dwReserved1 and **dwReserved2**
Reserved; set to zero.

DDVIDEOPORTSTATUS

The **DDVIDEOPORTSTATUS** structure describes the status of a video-port object. This structure is used with the **IDDVideoPortContainer::QueryVideoPortStatus** method.

```
typedef struct _DDVIDEOPORTSTATUS {
    DWORD dwSize;        // size of the structure
    BOOL  bInUse;        // TRUE if video port is currently being used
    DWORD dwFlags;       // not used
    DWORD dwReserved1;   // reserved for future use
    DDVIDEOPORTCONNECT VideoPortType;    // information about the connection
    DWORD dwReserved2;   // reserved for future use
    DWORD dwReserved3;   // reserved for future use
} DDVIDEOPORTSTATUS, *LPDDVIDEOPORTSTATUS;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before use.

bInUse
Value indicating the current status of the video port. This member is TRUE if the video port is currently being used, and FALSE otherwise.

dwFlags
Not currently used.

dwReserved1
Reserved; set to zero.

VideoPortType
A **DDVIDEOPORTCONNECT** structure that receives information about the video-port connection.

dwReserved2 and **dwReserved3**
Reserved; set to zero.

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all methods of the **IDirectDraw2**, **IDirectDrawSurface3**, **IDirectDrawPalette**, **IDirectDrawClipper** and **IDirectDrawVideoPort** interfaces. For a list of the error codes that each method can return, see the method description.

DD_OK

The request completed successfully.

DDERR_ALREADYINITIALIZED

The object has already been initialized.

DDERR_BLTFASTCANTCLIP

A DirectDrawClipper object is attached to a source surface that has passed into a call to the **IDirectDrawSurface3::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 3-D surfaces, or surfaces that are implicitly created, cannot be duplicated.

DDERR_CANTLOCKSURFACE

Access to this surface is refused because an attempt was made to lock the primary surface without DCI support.

DDERR_CANTPAGELOCK

An attempt to page lock a surface failed. Page lock will not work on a display-memory surface or an emulated primary surface.

DDERR_CANTPAGEUNLOCK

An attempt to page unlock a surface failed. Page unlock will not work on a display-memory surface or an emulated primary surface.

DDERR_CLIPPERISUSINGHWND

An attempt was made to set a clip list for a DirectDrawClipper object that is already monitoring a window handle.

DDERR_COLORKEYNOTSET

No source color key is specified for this operation.

DDERR_CURRENTLYNOTAVAIL

No support is currently available.

DDERR_DCALREADYCREATED

A device context (DC) has already been returned for this surface. Only one DC can be retrieved for each surface.

DDERR_DEVICEDOESNTOWNSURFACE

Surfaces created by one direct draw device cannot be used directly by another direct draw device.

DDERR_DIRECTDRAWALREADYCREATED

A DirectDraw object representing this driver has already been

created for this process.

DDERR_EXCEPTION

An exception was encountered while performing the requested operation.

DDERR_EXCLUSIVEMODEALREADYSET

An attempt was made to set the cooperative level when it was already set to exclusive.

DDERR_GENERIC

There is an undefined error condition.

DDERR_HEIGHTALIGN

The height of the provided rectangle is not a multiple of the required alignment.

DDERR_HWNDALREADYSET

The DirectDraw cooperative level window handle has already been set. It cannot be reset while the process has surfaces or palettes created.

DDERR_HWNDSUBCLASSED

DirectDraw is prevented from restoring state because the DirectDraw cooperative level window handle has been subclassed.

DDERR_IMPLICITLYCREATED

The surface cannot be restored because it is an implicitly created surface.

DDERR_INCOMPATIBLEPRIMARY

The primary surface creation request does not match with the existing primary surface.

DDERR_INVALIDCAPS

One or more of the capability bits passed to the callback function are incorrect.

DDERR_INVALIDCLIPLIST

DirectDraw does not support the provided clip list.

DDERR_INVALIDDIRECTDRAWGUID

The globally unique identifier (GUID) passed to the **DirectDrawCreate** function is not a valid DirectDraw driver identifier.

DDERR_INVALIDMODE

DirectDraw does not support the requested mode.

DDERR_INVALIDOBJECT

DirectDraw received a pointer that was an invalid DirectDraw object.

DDERR_INVALIDPARAMS

One or more of the parameters passed to the method are incorrect.

DDERR_INVALIDPIXELFORMAT

The pixel format was invalid as specified.

DDERR_INVALIDPOSITION

The position of the overlay on the destination is no longer legal.

DDERR_INVALIDRECT

The provided rectangle was invalid.

DDERR_INVALIDSURFACETYPE

The requested operation could not be performed because the surface was of the wrong type.

DDERR_LOCKEDSURFACES

One or more surfaces are locked, causing the failure of the requested operation.

DDERR_MOREDATA

There is more data available than the specified buffer size can hold.

DDERR_NO3D

No 3-D hardware or emulation is present.

DDERR_NOALPHAHW

No alpha acceleration hardware is present or available, causing the failure of the requested operation.

DDERR_NOBLTHW

No blitter hardware is present.

DDERR_NOCLIPLIST

No clip list is available.

DDERR_NOCLIPPERATTACHED

No DirectDrawClipper object is attached to the surface object.

DDERR_NOCOLORCONVHW

The operation cannot be carried out because no color-conversion hardware is present or available.

DDERR_NOCOLORKEY

The surface does not currently have a color key.

DDERR_NOCOLORKEYHW

The operation cannot be carried out because there is no hardware support for the destination color key.

DDERR_NOCOOPERATIVELEVELSET

A create function is called without the **IDirectDraw2::SetCooperativeLevel** method being called.

DDERR_NODC

No DC has ever been created for this surface.

DDERR_NODDROPSHW

No DirectDraw raster operation (ROP) hardware is available.

DDERR_NODIRECTDRAWHW

Hardware-only DirectDraw object creation is not possible; the driver does not support any hardware.

DDERR_NODIRECTDRAWSUPPORT

DirectDraw support is not possible with the current display driver.

DDERR_NOEMULATION

Software emulation is not available.

DDERR_NOEXCLUSIVEMODE

The operation requires the application to have exclusive mode, but the application does not have exclusive mode.

DDERR_NOFLIPHW

Flipping visible surfaces is not supported.

DDERR_NOGDI

No GDI is present.

DDERR_NOHWND

Clipper notification requires a window handle, or no window handle has been previously set as the cooperative level window handle.

DDERR_NOMIPMAPHW

The operation cannot be carried out because no mipmap texture mapping hardware is present or available.

DDERR_NOMIRRORHW

The operation cannot be carried out because no mirroring hardware is present or available.

DDERR_NONONLOCALVIDMEM

An attempt was made to allocate non-local video memory from a device that does not support non-local video memory.

DDERR_NOOPTIMIZEHW

The device does not support optimized surfaces.

DDERR_NOOVERLAYDEST

The **IDirectDrawSurface3::GetOverlayPosition** method is called on an overlay that the **IDirectDrawSurface3::UpdateOverlay** method has not been called on to establish a destination.

DDERR_NOOVERLAYHW

The operation cannot be carried out because no overlay hardware is present or available.

DDERR_NOPALETTEATTACHED

No palette object is attached to this surface.

DDERR_NOPALETTEHW

There is no hardware support for 16- or 256-color palettes.

DDERR_NORASTEROPHW

The operation cannot be carried out because no appropriate raster operation hardware is present or available.

DDERR_NOROTATIONHW

The operation cannot be carried out because no rotation hardware is present or available.

DDERR_NOSTRETCHHW

The operation cannot be carried out because there is no hardware support for stretching.

DDERR_NOT4BITCOLOR

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

DDERR_NOT4BITCOLORINDEX

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

DDERR_NOT8BITCOLOR

The DirectDrawSurface object is not using an 8-bit color palette and the requested operation requires an 8-bit color palette.

DDERR_NOTAOVERLAYSURFACE

An overlay component is called for a non-overlay surface.

DDERR_NOTEXTUREHW

The operation cannot be carried out because no texture-mapping hardware is present or available.

DDERR_NOTFLIPPABLE

An attempt has been made to flip a surface that cannot be flipped.

DDERR_NOTFOUND

The requested item was not found.

DDERR_NOTINITIALIZED

An attempt was made to call an interface method of a DirectDraw object created by **CoCreateInstance** before the object was initialized.

DDERR_NOTLOADED

The surface is an optimized surface, but it has not yet been allocated any memory.

DDERR_NOTLOCKED

An attempt is made to unlock a surface that was not locked.

DDERR_NOTPAGELOCKED

An attempt is made to page unlock a surface with no outstanding page locks.

DDERR_NOTPALETTIZED

The surface being used is not a palette-based surface.

DDERR_NOVSYNCHW

The operation cannot be carried out because there is no hardware support for vertical blank synchronized operations.

DDERR_NOZBUFFERHW

The operation to create a z-buffer in display memory or to perform a blit using a z-buffer cannot be carried out because there is no hardware support for z-buffers.

DDERR_NOZOVERLAYHW

The overlay surfaces cannot be z-layered based on the z-order because the hardware does not support z-ordering of overlays.

DDERR_OUTOFCAPS

The hardware needed for the requested operation has already been allocated.

DDERR_OUTOFMEMORY

DirectDraw does not have enough memory to perform the operation.

DDERR_OUTOFVIDEOMEMORY

DirectDraw does not have enough display memory to perform the operation.

DDERR_OVERLAYCANTCLIP

The hardware does not support clipped overlays.

DDERR_OVERLAYCOLORKEYONLYONEACTIVE

An attempt was made to have more than one color key active on an overlay.

DDERR_OVERLAYNOTVISIBLE

The **IDirectDrawSurface3::GetOverlayPosition** method is called on a hidden overlay.

DDERR_PALETTEBUSY

Access to this palette is refused because the palette is locked by another thread.

DDERR_PRIMARYSURFACEALREADYEXISTS

This process has already created a primary surface.

DDERR_REGIONTOOSMALL

The region passed to the **IDirectDrawClipper::GetClipList** method is too small.

DDERR_SURFACEALREADYATTACHED

An attempt was made to attach a surface to another surface to which it is already attached.

DDERR_SURFACEALREADYDEPENDENT

An attempt was made to make a surface a dependency of another surface to which it is already dependent.

DDERR_SURFACEBUSY

Access to the surface is refused because the surface is locked by another thread.

DDERR_SURFACEISOBSCURED

Access to the surface is refused because the surface is obscured.

DDERR_SURFACELOST

Access to the surface is refused because the surface memory is gone. The DirectDrawSurface object representing this surface should have the **IDirectDrawSurface3::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_VIDEONOTACTIVE

The video port is not active.

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.

Pixel Format Masks

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

- Texture Map Formats
- Off-Screen Surface Formats

Texture Map Formats

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

Pixel format flags	Bit depth	Masks
DDPF_RGB | 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 HEL can create these formats in system memory. The DirectDraw device driver for a 3-D-accelerated display card may create textures of other formats in display memory. Such a driver exports the DDSCAPS_TEXTURE flag to indicate that it can create textures.

Off-Screen Surface Formats

The following table shows the pixel formats for off-screen plain surfaces supported by the DirectX® 5 HEL. The Masks column contains the red, green, blue, and alpha masks for each set of pixel format flags and bit depths.

Pixel format flags	Bit depth	Masks
DDPF_RGB | DDPF_PALETTEINDEXED1	1	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED2	2	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED4	4	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB | DDPF_PALETTEINDEXED8	8	R: 0x00000000 G: 0x00000000 B: 0x00000000 A: 0x00000000
DDPF_RGB	16	R: 0x0000F800 G: 0x000007E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	16	R: 0x00007C00 G: 0x000003E0 B: 0x0000001F A: 0x00000000
DDPF_RGB	24	R: 0x00FF0000 G: 0x0000FF00

		B: 0x000000FF A: 0x00000000
DDPF_RGB	24	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000
DDPF_RGB	32	R: 0x00FF0000 G: 0x0000FF00 B: 0x000000FF A: 0x00000000
DDPF_RGB	32	R: 0x000000FF G: 0x0000FF00 B: 0x00FF0000 A: 0x00000000

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

Four Character Codes (FOURCC)

DirectDraw utilizes a special set of codes that are four characters in length. These codes, called four character codes or FOURCCs, are stored in file headers of files containing multimedia data such as bitmap images, sound, or video. FOURCCs describe the software technology that was used to produce multimedia data. By implication, they also describe the format of the data itself.

DirectDraw applications use FOURCCs for image color and format conversion. If an application calls the **IDirectDrawSurface3::GetPixelFormat** method to request the pixel format of a surface whose format is not RGB, the **dwFourCC** member of the **DDPIXELFORMAT** structure identifies the FOURCC when the method returns. For more information, see Converting Color and Format.

In addition, the **biCompression** member of the **BITMAPINFOHEADER** structure can be set to a FOURCC to indicate the codec used to compress or decompress an image.

FOURCCs are registered with Microsoft by the vendors of the respective multimedia software technologies. Some common FOURCCs appear in the list below.

FOURCC	Company	Technology Name
AUR2	AuraVision Corporation	AuraVision Aura 2: YUV 422
AURA	AuraVision Corporation	AuraVision Aura 1: YUV 411
CHAM	Winnov, Inc.	MM_WINNOV_CAVIARA_CHAMPAGNE
CVID	Supermac	Cinepak by Supermac
CYUV	Creative Labs, Inc	Creative Labs YUV
FVF1	Iterated Systems, Inc.	Fractal Video Frame
IF09	Intel Corporation	Intel Intermediate YUV9
IV31	Intel Corporation	Indeo 3.1
JPEG	Microsoft Corporation	Still Image JPEG DIB
MJPG	Microsoft Corporation	Motion JPEG Dib Format
MRLE	Microsoft Corporation	Run Length Encoding
MSVC	Microsoft Corporation	Video 1
PHMO	IBM Corporation	Photomotion
RT21	Intel Corporation	Indeo 2.1
ULTI	IBM Corporation	Ultimotion
V422	Vitec Multimedia	24 bit YUV 4:2:2
V655	Vitec Multimedia	16 bit YUV 4:2:2
VDCT	Vitec Multimedia	Video Maker Pro DIB
VIDS	Vitec Multimedia	YUV 4:2:2 CCIR 601 for V422
YU92	Intel Corporation	YUV
YUV8	Winnov, Inc.	MM_WINNOV_CAVIAR_YUV8
YUV9	Intel Corporation	YUV9
YUYV	Canopus, Co., Ltd.	BI_YUYV, Canopus
ZPEG	Metheus	Video Zipper

DirectSound

This section provides information about the DirectSound® component of DirectX®. Information is found under the following main headings:

- About DirectSound
- Why Use DirectSound?
- DirectSound Architecture
- DirectSound Essentials
- DirectSound Reference

About DirectSound

The Microsoft® DirectSound® application programming interface (API) is the audio component of the DirectX® Programmer's Reference of the Platform Software Development Kit (SDK). DirectSound provides low-latency mixing, hardware acceleration, and direct access to the sound device. It provides this functionality while maintaining compatibility with existing device drivers.

New in this release of DirectSound are capabilities for audio capture. Also new is support for property sets, which enable application developers to take advantage of extended services offered by sound cards and their associated drivers.

Why Use DirectSound?

The overriding design goal in DirectX is speed. Like other components of DirectX, DirectSound allows you to use the hardware in the most efficient way possible while insulating you from the specific details of that hardware with a device-independent interface. Your applications will work well with the simplest audio hardware but will also take advantage of the special features of cards and drivers that have been enhanced for use with DirectSound.

Here are some other things that DirectSound makes easy:

- Querying hardware capabilities at run time to determine the best solution for any given personal computer configuration
- Using property sets so that new hardware capabilities can be exploited even when they are not directly supported by DirectSound
- Low-latency mixing of audio streams for rapid response
- Developing 3-D sound
- Capturing sound

Despite the advantages of DirectSound, the standard waveform-audio functions in Windows® continue to be a practical solution for certain tasks. For example, an application can easily play a single sound or *audio stream*, such as introductory music, by using the **PlaySound** or **waveOut** functions.

DirectSound Architecture

This section introduces the components of DirectSound and explains how DirectSound works with the hardware and with other applications. The following topics are discussed:

- Architectural Overview
- Playback Overview
- Capture Overview
- Property Sets Overview
- Hardware Abstraction and Emulation
- System Integration

Architectural Overview

DirectSound implements a new model for playing and capturing digital sound samples and mixing sample sources. As with other object classes in the DirectX API, DirectSound uses the hardware to its greatest advantage whenever possible, and it emulates hardware features in software when the feature is not present in the hardware.

DirectSound playback is built on the **IDirectSound** component object model (COM) interface and on other interfaces for manipulating sound buffers and 3-D effects. These interfaces are **IDirectSoundBuffer**, **IDirectSound3DBuffer**, and **IDirectSound3DListener**

DirectSound capture is based on the **IDirectSoundCapture** and **IDirectSoundCaptureBuffer** COM interfaces.

Another COM interface, **IKsPropertySet**, provides methods that allow applications to take advantage of extended capabilities of sound cards.

Finally, the **IDirectSoundNotify** interface is used to signal events when playback or capture has reached a certain point in the buffer.

For more information about COM concepts that you should understand to create applications with the DirectX Programmer's Reference, see The Component Object Model.

Playback Overview

The DirectSound buffer object represents a buffer containing sound data in pulse code modulation (PCM) format. Buffer objects are used to start, stop, and pause sound playback, as well as to set attributes such as frequency and format.

The *primary sound buffer* holds the audio that the listener will hear. *Secondary sound buffers* each contain a single sound or stream of audio. DirectSound automatically creates a primary buffer, but it is the application's responsibility to create secondary buffers. When sounds in secondary buffers are played, DirectSound mixes them in the primary buffer and sends them to the output device. Only the available processing time limits the number of buffers that DirectSound can mix.

DirectSound does not include functions for parsing a sound file. It is your responsibility to stream data in the correct format into the secondary sound buffers.

Normally, buffers from only a single application are audible at any given moment, because only one application at a time has access to a particular DirectSound device.

Depending on the card type, DirectSound buffers can exist in hardware as on-board 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.

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. As a result, applications do not have to repeatedly play and stop their buffers when the input focus changes.

Through the **IDirectSoundNotify** interface, DirectSound provides a mechanism to notify the client when the play cursor reaches positions within a buffer that have been specified by the client, or when playback has stopped.

Capture Overview

The **DirectSoundCapture** object is used to query the capabilities of sound capture devices and to create buffers for capturing audio from an input source.

Audio capture functions already exist in Win32. The first release of **DirectSoundCapture** in the DirectX 5 Programmer's Reference does not provide any performance improvement over the existing **waveIn** functions. However, the **DirectSoundCapture** API allows application developers to create titles using consistent interfaces for both audio playback and capture. It also allows titles to be developed today that will benefit from new, improved driver models and API implementations in the future.

DirectSoundCapture allows capturing of compressed formats. In this first version, the underlying **waveIn** functions or the hardware provide support for compressed formats. **DirectSoundCapture** does not call the audio compression manager (ACM) functions itself.

The DirectSoundCaptureBuffer object represents a buffer used for capturing data from the input device. This buffer is circular; that is, when the input pointer reaches the end of the buffer, it starts again at the beginning.

The methods of the DirectSoundCaptureBuffer object allow you to retrieve the properties of the buffer, start and stop audio capture, and lock portions of the memory so that you can safely retrieve data for saving to a file or for some other purpose.

As with playback, DirectSound allows you to request notification when captured data reaches a specified position within the buffer, or when capture has stopped. This service is provided through the **IDirectSoundNotify** interface.

Property Sets Overview

Through property sets, DirectSound is able to support extended services offered by manufacturers of sound cards and their associated drivers.

Hardware vendors define new capabilities as properties and publish the specification for these properties. You, the application developer, can then use the methods of the **IKsPropertySet** interface to determine whether a particular set of properties is available on the target hardware and to manipulate those properties, for instance by turning special effects on and off.

Property sets allow for the unlimited extension of the capabilities of DirectSound. You use a single method, **IKsPropertySet::Set**, to alter the state of the device in any way specified by the manufacturer.

Hardware Abstraction and Emulation

DirectSound accesses the sound hardware through the DirectSound hardware-abstraction layer (HAL), an interface that is implemented by the audio-device driver.

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
- Causes the operation request to report failure when hardware is unavailable

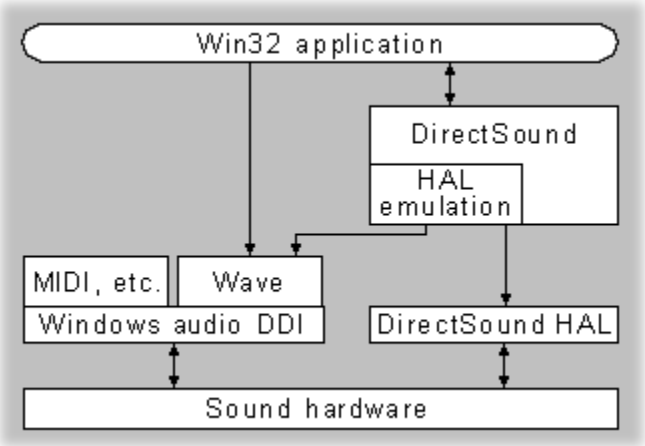
The device driver does not perform any software emulation; it simply reports the capabilities of the hardware to DirectSound and passes requests from DirectSound to the hardware. If the hardware cannot perform a requested operation, the device driver reports failure of the request and DirectSound emulates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

Your application can use DirectSound as long as the DirectX run-time files are present on the user's system. If the sound hardware does not have an installed DirectSound driver, DirectSound uses its hardware-emulation layer (HEL), which employs the Windows multimedia waveform-audio (**waveIn** and **waveOut**) functions. Most DirectSound features are still available through the HEL, but of course hardware acceleration is not possible.

DirectSound automatically takes advantage of accelerated sound hardware, including hardware mixing and hardware sound-buffer memory. Your application need not 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 at run time to receive a full description of the capabilities of the sound device, and then use different routines optimized for the presence or absence of a given feature. You can also specify which sound buffers should receive hardware acceleration.

System Integration

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



DirectSound and the standard Windows 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, an attempt to allocate that same device by using DirectSound will fail. Similarly, if a DirectSound driver has allocated a device, an attempt to allocate the device by using the waveform-audio driver will fail.

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

Note Microsoft Video for Windows currently uses the waveform-audio functions to play the audio track of an audio visual interleaved (.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 **IDirectSound::Release** before playing an .avi file. Applications can then re-create and reinitialize the DirectSound object and its DirectSoundBuffer objects when the video finishes playing.

DirectSound Essentials

This section gives a practical overview of how the various DirectSound interfaces are used in order to play and capture sound. The following topics are discussed:

- DirectSound Devices
- DirectSound Buffers
- Introduction to 3-D Sound
- DirectSound 3-D Buffers
- DirectSound 3-D Listeners
- DirectSoundCapture
- DirectSound Property Sets
- Optimizing DirectSound Performance

Note Most of the examples of method calls throughout this section are given in the C form, which accesses methods by means of a pointer to a table of pointers to functions, and requires a this pointer as the first parameter in any call. For example, a C call to the **IDirectSound::GetCaps** method takes this form:

```
lpDirectSound->lpVtbl->GetCaps(lpDirectSound, &dscaps);
```

You can simplify C calls to any of the DirectSound methods by using the macros defined in Dsound.h. For example:

```
IDirectSound_GetCaps(lpDirectSound, &dscaps);
```

The same method call in the C++ form, which treats COM interface methods just like class methods, looks like this:

```
lpDirectSound->GetCaps(&dscaps);
```

DirectSound Devices

The first step in implementing DirectSound in an application is to create a DirectSound object, which creates an instance of the **IDirectSound** interface.

A DirectSound object describes the audio hardware on a system. The **IDirectSound** interface enables your application to define and control the sound card, speaker, and memory environment.

This section describes how your application can enumerate available sound devices, create the DirectSound object for a device, and use the methods of the object to set the cooperative level, retrieve the capabilities of the device, create sound buffers, set the configuration of the system's speakers, and compact hardware memory.

Enumeration of Sound Devices

For an application that is simply going to play sounds through the user's preferred playback device, you need not enumerate the available devices. When you create the DirectSound object with NULL as the device identifier, the interface will automatically be associated with the default device if one is present. If no device driver is present, the call to the **DirectSoundCreate** function will fail.

However, if you are looking for a particular kind of device or need to work with two or more devices, you must get DirectSound to enumerate the devices available on the system.

Enumeration serves three purposes:

- Reports what hardware is available
- Supplies a GUID for each device
- Allows you to initialize DirectSound for each device as it is enumerated

To enumerate devices you must first set up a callback function that will be called each time DirectSound finds a device. You can do anything you want within this function, and you can give it any name, but you must declare it in the same form as **DSEnumCallback**, a prototype in this documentation. The callback function must return TRUE if enumeration is to continue, or FALSE otherwise (for instance, after finding a device with the capabilities you need).

If you are working with more than one device—for example, a capture and a playback device—the callback function is a good place to create and initialize the DirectSound object for each device.

The following example, extracted from Dsenum.c in the Dsshow sample, enumerates the available devices and adds information about each to a list in a combo box. Here is the callback function in its entirety:

```
BOOL CALLBACK DSEnumProc(LPGUID lpGUID,
                         LPCTSTR lpszDesc,
                         LPCTSTR lpszDrvName,
                         LPVOID lpContext )
    {
    HWND   hCombo = *(HWND *)lpContext;
    LPGUID lpTemp = NULL;

    if( lpGUID != NULL )
        {
        if(( lpTemp = LocalAlloc( LPTR, sizeof(GUID))) == NULL )
        return( TRUE );

        memcpy( lpTemp, lpGUID, sizeof(GUID));
    }

    ComboBox_AddString( hCombo, lpszDesc );
    ComboBox_SetItemData( hCombo,
                ComboBox_FindString( hCombo, 0, lpszDesc ),
                lpTemp );
    return( TRUE );
    }
```

The enumeration is set in motion when the dialog containing the combo box is initialized:

```
if (DirectSoundEnumerate((LPDSENUMCALLBACK)DSEnumProc, &hCombo)
    != DS_OK )
    {
    EndDialog( hDlg, TRUE );
```

```
    return( TRUE );
    }
```

Note that the address of the handle to the combo box is passed into **DirectSoundEnumerate**, which in turn passes it to the callback function. This parameter can be any 32-bit value that you want to have access to within the callback.

Creating the DirectSound Object

The simplest way to create the DirectSound object is with the **DirectSoundCreate** function. The first parameter of this function specifies the GUID of the device to be associated with the object. You can obtain this GUID by Enumeration of Sound Devices, or you can simply pass NULL to create the object for the default device.

```
LPDIRECTSOUND lpDirectSound;
HRESULT       hr;
hr = DirectSoundCreate(NULL, &lpDirectSound, NULL));
```

The function returns an error if there is no sound device or if the sound device is being used by the waveform-audio (non-DirectSound) functions. You should prepare your applications for this call to fail so that they can either continue without sound or prompt the user to close the application that is already using the sound device.

You can also create the DirectSound object by using the **CoCreateInstance** function, as follows:

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

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

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

   ```
   dsrval = CoCreateInstance(&CLSID_DirectSound,
                             NULL,
                             CLSCTX_INPROC_SERVER,
                             &IID_IDirectSound,
                             &lpds);
       if (SUCCEEDED(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. The *lpds* parameter is the uninitialized object **CoCreateInstance** returns.

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

Before you close the application, shut down COM by calling the **CoUninitialize** function, as follows:

```
CoUninitialize();
```

Cooperative Levels

Because Windows is a multitasking environment, more than one application may be working with a device driver at any one time. Through the use of cooperative levels, DirectX makes sure that each application does not gain access to the device in the wrong way or at the wrong time. Each DirectSound application has a cooperative level that determines the extent to which it is allowed to access the device.

After creating a DirectSound object, you must set the cooperative level for the device with the **IDirectSound::SetCooperativeLevel** method before you can play sounds.

The following example sets the cooperative level for the DirectSound device initialized at Creating the DirectSound Object. The *hwnd* parameter is the handle to the application window.

```
HRESULT hr = lpDirectSound->lpVtbl->SetCooperativeLevel(
                      lpDirectSound, hwnd, DSSCL_NORMAL);
```

DirectSound defines four cooperative levels for sound devices: normal, priority, exclusive, and write-primary. Most applications will use the sound device at the normal cooperative level, which allows for orderly switching between applications that use the sound card.

Normal Cooperative Level

At the normal cooperative level, the application cannot set the format of the primary sound buffer, write to the primary buffer, or compact the on-board memory of the device. All applications at this cooperative level use a primary buffer format of 22 kHz, stereo sound, and 8-bit samples, so that the device can switch between applications as smoothly as possible.

Priority Cooperative Level

When using a DirectSound device with the priority cooperative level, the application has first rights to hardware resources, such as hardware mixing, and can set the format of the primary sound buffer and compact the on-board memory of the device.

Exclusive Cooperative Level

At the exclusive cooperative level, the application has all the privileges of the priority level. In addition, when the application is in the foreground, its buffers are the only ones that are audible.

Write-primary Cooperative Level

The highest cooperative level is write-primary. When using a DirectSound device with this cooperative level, your application has direct access to the primary sound buffer. In this mode, the application must write directly to the primary buffer. Secondary buffers cannot be played while this is going on.

An application must be set to the write-primary level in order to obtain direct write access to the audio samples in the primary buffer. If the application is not set to this level, then all calls to the **IDirectSoundBuffer::Lock** method for the primary buffer will fail.

When your application is set to the write-primary cooperative level and gains the foreground, all secondary buffers for other applications are stopped and marked as lost. When your application in turn moves to the background, its primary buffer is marked as lost and must be restored when the application again moves to the foreground. For more information, see Buffer Management.

You cannot set the write-primary cooperative level if a DirectSound driver is not present on the user's system. To determine whether this is the case, call the **IDirectSound::GetCaps** method and check for the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

For more information, see Access to the Primary Buffer.

Device Capabilities

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

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.

The following example retrieves the capabilities of the device that was initialized in Creating the DirectSound Object.

```
DSCAPS dscaps;

dscaps.dwSize = sizeof(DSCAPS);
HRESULT hr = lpDirectSound->lpVtbl->GetCaps(lpDirectSound,
    &dscaps);
```

The **DSCAPS** structure receives information about the performance and resources of the sound device, including the maximum resources of each type and the resources that are currently available. Note that the **dwSize** member of this structure must be initialized before the method is called.

It is unwise to make assumptions about the behavior of the sound device; if you do, your application might work on some sound devices but not on others. Furthermore, future devices might behave differently.

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 to create the next buffer.

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**.

Compacting Hardware Memory

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

DirectSound Buffers

This section covers the creation and management of DirectSoundBuffer objects, which are the fundamental mechanism for playing sounds. The following topics are discussed:

- Buffer Basics
- Static and Streaming Sound Buffers
- Creating Secondary Buffers
- Buffer Control Options
- Access to the Primary Buffer
- Playing Sounds
- Playback Controls
- Current Play and Write Positions
- Play Buffer Notification
- Mixing Sounds
- Custom Mixers
- Buffer Management
- Compressed Wave Formats

Most of the information in this section applies to 3-D sound buffers as well. For information specific to the **IDirectSound3DBuffer** interface, see DirectSound 3-D Buffers.

For information about capture buffers, see **DirectSoundCapture**.

Buffer Basics

When you initialize DirectSound in your application, it automatically creates and manages a primary sound buffer for mixing sounds and sending them to the output device.

Your application must create at least one secondary sound buffer for storing and playing individual sounds. For more information on how to do this, see Creating Secondary Buffers.

A secondary buffer can exist throughout the life of an application or it may be destroyed when no longer needed. It may contain a single sound that is to be played repeatedly, such as a sound effect in a game, or it may be filled with new data from time to time. The application can play a sound stored in a secondary buffer as a single event or as a looping sound that plays continuously.

Secondary buffers can also be used to stream data, in cases where a sound file contains more data than can conveniently be stored in memory.

For more information on the different kinds of secondary buffers, see Static and Streaming Sound Buffers.

You can create two or more secondary buffers in the same physical memory by using the **IDirectSound::DuplicateSoundBuffer** method, provided the original buffer is not on the sound hardware.

You can mix sounds from different secondary buffers simply by playing them at the same time. Data from secondary buffers is mixed by DirectSound in the primary buffer. Any number of secondary buffers can be played at one time, up to the limits of processing power.

The DirectSound mixer can provide as little as 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 at the same time. However, if DirectSound must emulate hardware features in software, the mixer cannot achieve low latency and a longer delay (typically 100-150 milliseconds) occurs before the sound is reproduced.

Normally you do not have to concern yourself at all with the primary buffer; DirectSound manages it behind the scenes. However, if your application is to perform its own mixing, DirectSound will let you write directly to the primary buffer. If you do this, you cannot also use secondary buffers. For more information, see Access to the Primary Buffer.

Static and Streaming Sound Buffers

When you create a secondary sound buffer, you specify whether it is a *static sound buffer* or a *streaming sound buffer*. A static buffer contains a complete sound in memory. A streaming buffer holds only a portion of a sound, such as 3 seconds of data from a 15-second bit of voice dialog. When using a streaming sound buffer, your application must periodically write new data to the buffer.

If a sound device has on-board sound memory, DirectSound attempts to place static buffers in the hardware memory. These buffers can then take advantage of hardware mixing, and the processing system incurs little or no overhead to mix these sounds. This is particularly useful for sounds your application plays repeatedly, because the sound data must be downloaded only once to the hardware memory.

Streaming buffers are generally located in main system memory to allow efficient writing to the buffer, although you can use hardware mixing on peripheral component interconnect (PCI) machines or other fast buses.

DirectSound distinguishes between static and streaming buffers in order to optimize performance, but it does not restrict how you can use the buffer. If a streaming buffer is big enough, there is nothing to prevent you from writing an entire sound to it in one chunk. In fact, if you do not intend to use the sound more than once, it can be more efficient to use a streaming buffer because by doing so you eliminate the step of downloading the data to hardware memory.

Your application may attempt to explicitly locate buffers in hardware or software. If you attempt to create a hardware buffer and there is insufficient memory or mixing capacity, the buffer creation request fails. Many existing sound cards do not have any on-board memory or mixing capacity, so no hardware buffers can be created on these devices.

For more information, see Creating Secondary Buffers.

Creating Secondary Buffers

To create a sound buffer, your application fills a **DSBUFFERDESC** structure and then calls the **IDirectSound::CreateSoundBuffer** method. This method creates a DirectSoundBuffer object and returns a pointer to an **IDirectSoundBuffer** interface. Your application uses this interface to manipulate and play the buffer.

The following example illustrates how to create a basic secondary sound 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);
    // Need default controls (pan, volume, frequency).
    dsbdesc.dwFlags = DSBCAPS_CTRLDEFAULT;
    // 3-second buffer.
    dsbdesc.dwBufferBytes = 3 * pcmwf.wf.nAvgBytesPerSec;
    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;
    }
}
```

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

If your application must explicitly locate buffers in hardware or software, you can specify either the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag in the **DSBUFFERDESC** structure. If the DSBCAPS_LOCHARDWARE flag is specified and there is insufficient hardware memory or mixing capacity, the buffer creation request fails.

You can ascertain the location of an existing buffer by using 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 is always specified.

When you create a sound buffer, you can indicate that a buffer is static by specifying the DSBCAPS_STATIC flag. If you do not specify this flag, the buffer is a streaming buffer. For more information, see Static and Streaming Sound Buffers.

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

Buffer Control Options

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

To obtain the best performance on all sound cards, your application should specify only control options it will 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 the **IDirectSoundBuffer::SetVolume** method, the method succeeds 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.

Access to the Primary Buffer

For applications that require specialized mixing or other effects not supported by secondary buffers, DirectSound allows direct access to the primary buffer.

When you obtain write access to a primary sound buffer, other DirectSound features become unavailable. Secondary buffers are not mixed and, consequently, hardware-accelerated mixing is unavailable.

Most 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 an 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.

You cannot specify the size of the primary buffer, and you must accept the returned size after the buffer is created. 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 to prevent the previously written data from being replayed.

You create an accessible primary buffer by specifying the DSBCAPS_PRIMARYBUFFER flag in the **DSBUFFERDESC** structure passed to the **IDirectSound::CreateSoundBuffer** method. If you want to write to the buffer, the cooperative level must be DSSCL_WRITEPRIMARY.

Primary sound buffers must be played with looping. Ensure 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;
    // Buffer size is determined by sound hardware.
    dsbdesc.dwBufferBytes = 0;
    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;
            }
        }
    }
    // SetCooperativeLevel failed.
    // CreateSoundBuffer, or SetFormat.
    *lplpDsb = NULL;
    *lpdwBufferSize = 0;
    return FALSE;
}
```

You cannot obtain write access to a primary buffer unless it exists in hardware. To determine whether this is the case, call the **IDirectSoundBuffer::GetCaps** method and check for the DSBCAPS_LOCHARDWARE flag in the **dwFlags** member of the **DSBCAPS** structure that is returned. If you attempt to lock a primary buffer that is emulated in software, the call will fail.

You may also create a primary buffer object without write access, by specifying a cooperative level other than DSSCL_WRITEPRIMARY. One reason for doing this would be to call the **IDirectSoundBuffer::Play** method for the primary buffer, in order to eliminate problems associated with frequent short periods of silence. For more information, see Playing the Primary Buffer Continuously.

See also Custom Mixers.

Playing Sounds

Playing a sound consists of the following steps:

1. Lock a portion of the secondary buffer (**IDirectSoundBuffer::Lock**). This method returns a pointer to the address where writing will begin, based on the offset from the beginning of the buffer that you pass in.
2. Write the audio data to the buffer.
3. Unlock the buffer (**IDirectSoundBuffer::Unlock**).
4. Send the sound to the primary buffer and from there to the output device (**IDirectSoundBuffer::Play**).

Because streaming sound buffers usually play continually and are conceptually circular, DirectSound returns two write pointers when locking a sound buffer. For example, if you tried to lock 300 bytes beginning at the midpoint of a 400-byte buffer, the **Lock** method would return one pointer to the last 200 bytes of the buffer, and a second pointer to the first 100 bytes. The second pointer is NULL if the locked portion of the buffer does not wrap around.

Normally the buffer stops playing automatically when the end is reached. However, if the DSBPLAY_LOOPING flag was set in the *dwFlags* parameter to the **Play** method, the buffer will play repeatedly until the application calls the **IDirectSoundBuffer::Stop** method, at which point the play cursor is moved to the beginning of the buffer.

For streaming sound buffers, your application is responsible for ensuring that the next block of data is written to the buffer before the current play position loops back to the beginning. (For more on the play position, see Current Play and Write Positions.) You can do this by setting notification positions so that an event is signaled whenever the current play position reaches a certain point. Applications should write at least 1 second ahead of the current play position to minimize the possibility of gaps in the audio output during playback.

The following C example writes data to a sound buffer, starting at the offset into the buffer passed in *dwOffset*:

```
BOOL AppWriteDataToBuffer(
    LPDIRECTSOUNDBUFFER lpDsb,  // the DirectSound buffer
    DWORD dwOffset,             // our own write cursor
    LPBYTE lpbSoundData,        // start of our data
    DWORD dwSoundBytes)         // size of block to copy
{
    LPVOID lpvPtr1;
    DWORD dwBytes1;
    LPVOID lpvPtr2;
    DWORD dwBytes2;
    HRESULT hr;
    // Obtain memory address of write block. This will be in two parts
    // if the block wraps around.
    hr = lpDsb->lpVtbl->Lock(lpDsb, dwOffset, dwSoundBytes, &lpvPtr1,
        &dwBytes1, &lpvPtr2, &dwBytes2, 0);

    // If DSERR_BUFFERLOST is returned, 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;
        }
    }
    // Lock, Unlock, or Restore failed.
    return FALSE;
}
```

Playback Controls

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 play. You cannot change the frequency of the primary buffer.

To retrieve and set the pan, you can call the **IDirectSoundBuffer::GetPan** and **IDirectSoundBuffer::SetPan** methods. You cannot change the pan of the primary buffer.

Current Play and Write Positions

DirectSound maintains two pointers into the buffer: the current play position (or play cursor) and the current write position (or write cursor). These positions are byte offsets into the buffer, not absolute memory addresses.

The **IDirectSoundBuffer::Play** method always starts playing at the buffer's current play position. When a buffer is created, the play position is set to zero. As a sound is played, the play position moves and always points to the next byte of data to be output. When the buffer is stopped, the play position remains where it is.

The current write position is the point after which it is safe to write data into the buffer. The block between the current play position and the current write position is already committed to be played, and cannot be changed safely.

Visualize the buffer as a clock face, with data written to it in a clockwise direction. The play position and the write position are like two hands sweeping around the face at the same speed, the write position always keeping a little ahead of the play position. If the play position points to the 1 and the write position points to the 2, it is only safe to write data after the 2. Data between the 1 and the 2 may already have been queued for playback by DirectSound and should not be touched.

Note The write position moves with the play position, not with data written to the buffer. If you're streaming data, you are responsible for maintaining your own pointer into the buffer to indicate where the next block of data should be written.

Also note that the *dwWriteCursor* parameter to the **IDirectSoundBuffer::Lock** method is not the current write position; it is the offset within the buffer where you actually intend to begin writing data. (If you do want to begin writing at the current write position, you specify DSBLOCK_FROMWRITECURSOR in the *dwFlags* parameter. In this case the *dwWriteCursor* parameter is ignored.)

An application can retrieve the current play and write positions by calling the **IDirectSoundBuffer::GetCurrentPosition** method. The **IDirectSoundBuffer::SetCurrentPosition** method lets you set the current play position, but the current write position cannot be changed.

Play Buffer Notification

Particularly when streaming audio, you may want your application to be notified when the play cursor reaches a certain point in the buffer, or when playback is stopped. With the **IDirectSoundNotify::SetNotificationPositions** method you can set any number of points within the buffer where events are to be signaled. You cannot do this while the buffer is playing.

First you have to obtain a pointer to the **IDirectSoundNotify** interface. You can do this with the buffer object's **QueryInterface** method, as in the following C++ example:

```
// LPDIRECTSOUNDBUFFER lpDsbSecondary;
// The buffer has been initialized already.
LPDIRECTSOUNDNOTIFY lpDsNotify;  // pointer to the interface

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSoundNotify,
                                            &lpDsNotify);
if (SUCCEEDED(hr))
  {
  // Go ahead and use lpDsNotify->SetNotificationPositions.
  }
```

Note The **IDirectSoundNotify** interface is associated with the object that obtained the pointer, in this case the secondary buffer. The methods of the new interface will automatically apply to that buffer.

Now create an event object with the Win32 **CreateEvent** function. You put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure, and in the **dwOffset** member of that structure you specify the offset within the buffer where you want the event to be signaled. Then you pass the address of the structure—or of an array of structures, if you want to set more than one notification position—to the **IDirectSoundNotify::SetNotificationPositions** method.

The following example sets a single notification position. The event will be signaled when playback stops, either because it was not looping and the end of the buffer has been reached, or because the application called the **IDirectSoundBuffer::Stop** method.

```
DSBPOSITIONNOTIFY PositionNotify;

PositionNotify.Offset = DSBPN_OFFSETSTOP;
PositionNotify.hEventNotify = hMyEvent;
// hMyEvent is the handle returned by CreateEvent()

lpDsNotify->SetNotificationPositions(1, &PositionNotify);
```

Mixing Sounds

It is easy to mix multiple streams with DirectSound. You simply create secondary sound buffers, receiving an **IDirectSoundBuffer** interface for each sound. You then play the buffers simultaneously. DirectSound takes care of the mixing in the primary sound buffer and plays the result.

The DirectSound mixer can obtain the best results from 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 downloads these buffers to the sound hardware memory, where available, and consequently does not incur any processing overhead in mixing these buffers. The most important static sound buffers should be created first to give them first priority for hardware acceleration.

The DirectSound mixer produces the best sound quality if all 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.

Your application can change the hardware output format by creating a primary sound buffer and calling the **IDirectSoundBuffer::SetFormat** method. Note that this primary buffer is for control purposes only; creating it is not the same as obtaining write access to the primary buffer as described under Access to the Primary Buffer, and you do not need the DSSCL_WRITEPRIMARY cooperative level. However, you do need a cooperative level of DSSCL_PRIORITY or higher in order to call the **SetFormat** method. DirectSound will restore the hardware format to the format specified in the last call every time the application gains the input focus.

Custom Mixers

Most applications will use the DirectSound mixer; it should be sufficient for almost all mixing needs and it automatically takes advantage of any available hardware acceleration. However, if an application requires some other functionality that DirectSound does not provide, it can obtain write access to the primary sound buffer and mix streams directly into it.

To implement a custom mixer, the application must first obtain the DSSCL_WRITEPRIMARY cooperative level and then create a primary sound buffer. (See Access to the Primary Buffer.) It can then lock the buffer, write data to it, unlock it, and play it just like any other buffer. (See Playing Sounds.) Note however 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. The **AppMixIntoPrimaryBuffer** 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 that mixes several streams together, as specified in the application-defined **AppStreamInfo** structure, and writes the result to the specified pointer.

```
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 DSERR_BUFFERLOST is returned, 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 it wrapped around.
        }
        // Release the data back to DirectSound.
        hr = lpDsbPrimary->lpVtbl->Unlock(lpDsbPrimary,
                                          lpvPtr1, dwBytes1,
                                          lpvPtr2, dwBytes2);
```

```
        if (DS_OK == hr) {
            // Success.
            return TRUE;
        }
    }
    // Lock or Unlock failed.
    return FALSE;
}
```

Buffer Management

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. You also can use the **IDirectSoundBuffer::GetFormat** and **IDirectSoundBuffer::SetFormat** methods to retrieve and set the format of the sound data in the primary sound buffer.

Note After a secondary sound buffer is created, its format is fixed. If you need a secondary buffer that uses another format, you must create a new sound buffer with this format.

Memory for a sound buffer can be lost in certain situations. In particular, this can occur when buffers are located in the hardware sound memory. In the worst case, the sound card itself might be removed from the system while in use; this situation can occur with PCMCIA sound cards.

Loss can also occur when an application with the write-primary cooperative level moves to the foreground. If this flag is set, DirectSound makes all other sound buffers lost so that the foreground application can write directly to the primary buffer. The DSERR_BUFFERLOST error code is returned when the **IDirectSoundBuffer::Lock** or **IDirectSoundBuffer::Play** method is called for any other buffer. When the application lowers its cooperative level from write-primary, or moves to the background, 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.

Compressed Wave Formats

DirectSound does not currently support compressed wave formats. Applications should use the audio compression manager (ACM) functions, provided with the Win32 APIs in the Platform SDK, to convert compressed audio to pulse-code 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 to the sound buffer for maximum efficiency.

Introduction to 3-D Sound

DirectSound enables an application to change the apparent position and movement of a sound source. A sound source can be a point from which sounds radiate in all directions or a cone outside which sounds are attenuated. Applications can also modify sounds using Doppler shift.

Although these effects are audible using standard loudspeakers, they are more obvious and compelling when the user wears headphones.

This overview introduces the basic concepts of 3-D sound as implemented by DirectSound. The following topics are discussed:

- Perception of Sound Positions
- Listeners
- Sound Cones
- Distance Measurements
- Doppler Shift
- Integration with Direct3D
- Mono and Stereo Sources

Specific information on how to use 3-D sound in an application is found in the following sections:

- DirectSound 3-D Buffers
- DirectSound 3-D Listeners

Perception of Sound Positions

In the real world, the perception of a sound's position in space is influenced by a number of factors, including the following:

- *Volume.* The farther an object is from the listener, the quieter it sounds. This phenomenon is known as rolloff.
- *Arrival offset.* A sound emitted by a source to the listener's right will arrive at the right ear slightly before it arrives at the left ear. (The duration of this offset is approximately a millisecond.)
- *Muffling.* The orientation of the ears ensures that sounds coming from behind the listener are slightly muffled compared with sounds coming from in front. In addition, if a sound is coming from the right, the sounds reaching the left ear will be muffled by the mass of the listener's head as well as by the orientation of the left ear.

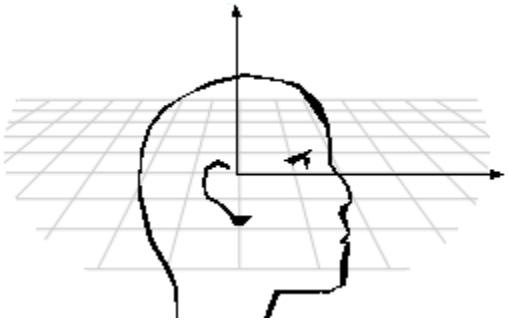
Although these are not the only cues people use to discern the position of sound, they are the main ones, and they are the factors that have been implemented in the positioning system of DirectSound. When hardware that supports 3-D sound becomes generally available, other positioning cues might be incorporated into the system, including the difference in how high- and low-frequency sounds are muffled by the mass of the listener's head and the reflections of sound off the shoulders and earlobes.

One of the most important sound-positioning cues is the apparent visual position of the sound source. If a projectile appears as a dot in the distance and grows to the size of an intercontinental missile before it roars past the viewer's head, the listener does not need subtle acoustical cues in order to perceive that the sound has gone past.

Listeners

Listeners experience an identical sonic effect when an object moves in a 90-degree arc around them or if they move their heads 90 degrees relative to the object. Programmatically, however, it is often much simpler to change the position or orientation of the listener than to change the position of every other object in a scene. DirectSound makes this possible through the **IDirectSound3DListener** interface.

Listener *orientation* is defined by the relationship between two vectors that share an origin: the *top* and *front vectors*. The top vector originates from the center of the listener's head and points straight up through the top of the head. The front vector also originates from the center of the listener's face. The following illustration shows the directions of these vectors:

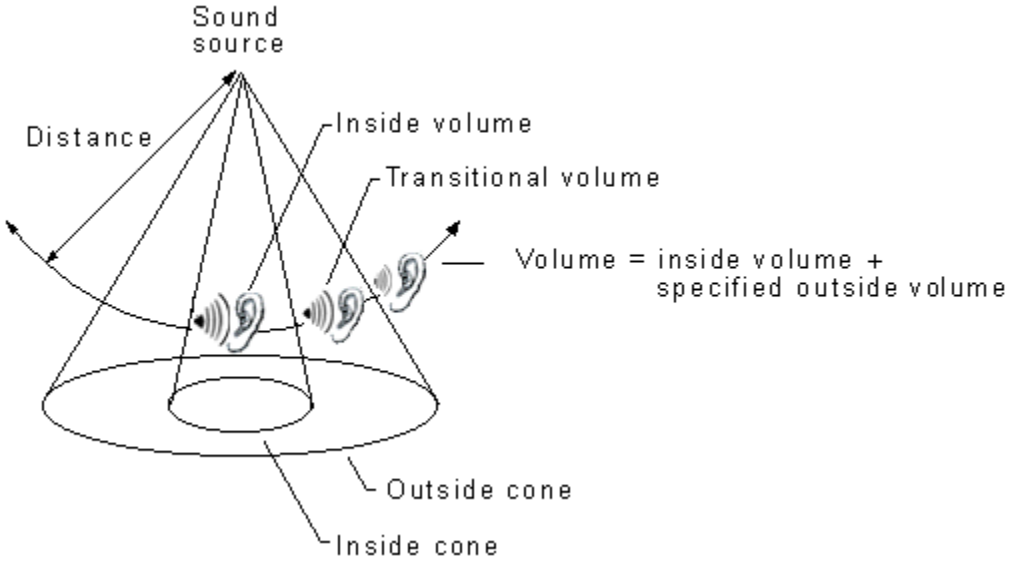


Sound Cones

A sound with a position but no orientation is a point source; the farther the listener is from the sound, in any direction, the quieter the sound. A sound with a position and an orientation is a sound cone.

In DirectSound, sound cones include an inside cone and an outside cone. Within the inside cone, the volume is at the maximum level for that sound source. (Because DirectSound does not support amplification, the maximum volume level is zero; all other volume levels are negative values that represent an attenuation of the maximum volume.) Outside the outside cone, the volume is the specified outside volume added to the inside volume. If an application sets the outside volume to DSBVOLUME_MIN, for example, the sound source will be inaudible outside the outside cone. Between the outside and inside cones, the volume changes gradually from one level to the other.

The concept of sound cones is shown in the following illustration:



Technically, every sound buffer represented by the **IDirectSound3DBuffer** interface is a sound cone, but often these sound cones behave like omnidirectional sound sources. For example, the default value for the volume outside the sound cone is zero; unless the application changes this value, the volume will be the same inside and outside the cone, and sound will not have any apparent orientation. You could also make the sound-cone angles as wide as you want, effectively making the sound cone a sphere.

Designing sound cones properly can add dramatic effects to your application. For example, you could position the sound source in the center of a room, setting its orientation toward a door. Then set the angle of the outside cone so that it extends to the width of the doorway and set the outside cone volume to inaudible. The user, when passing the open door, will suddenly hear the voice emanating from the room.

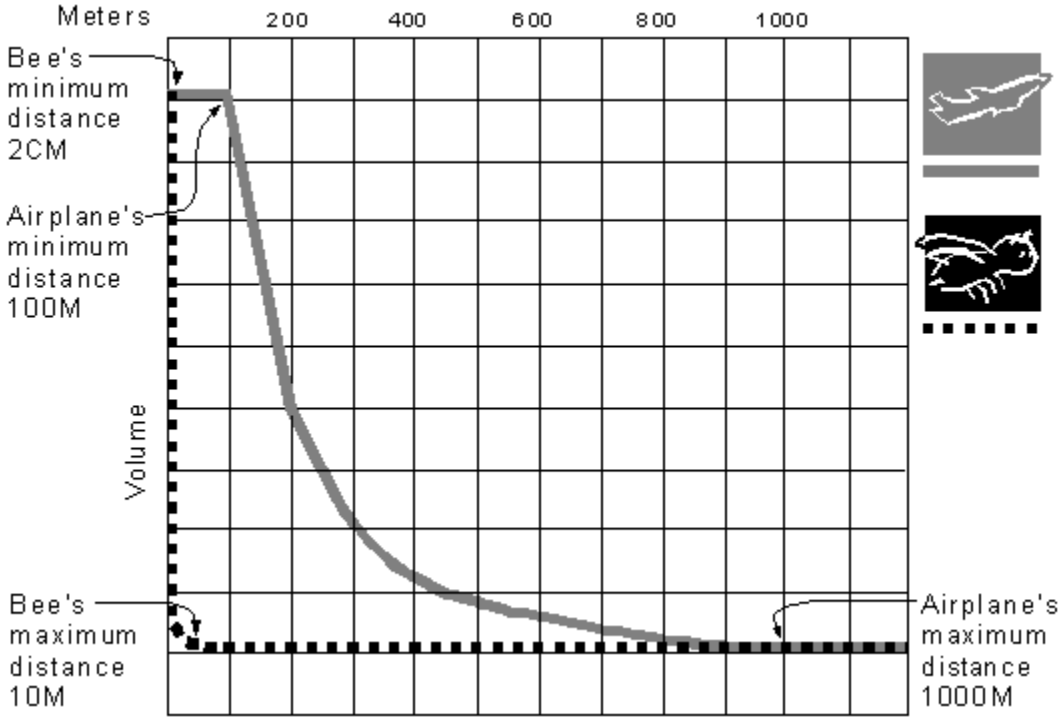
Distance Measurements

The 3-D effects of DirectSound use meters as the default unit of distance measurements. If your application does not use meters, it need not convert between units of measure to maintain compatibility with the component. Instead, the application can set a *distance factor*, which is a floating-point value representing meters per application-specified distance unit. For example, if your application uses feet as its unit of measure, it could specify a distance factor of .30480006096, which is the number of meters in a foot.

The default distance measurements for the 3-D sound effects mimic the natural world. Many application designers choose to change these values, however, to make the effects more dramatic. Exaggerated Doppler effects or exaggerated sound attenuation with distance can make an application more exciting.

As a listener approaches a sound source, the sound gets louder. Past a certain point, however, it is not reasonable for the volume to continue to increase; either the maximum (zero) has been reached, or the nature of the sound source imposes a logical limit. This is the *minimum distance* for the sound source. Similarly, the *maximum distance* for a sound source is the distance beyond which the sound does not get any quieter.

The minimum distance is especially useful when an application must compensate for the difference in absolute volume levels of different sounds. Although a jet engine is much louder than a bee, for example, for practical reasons these sounds must be recorded at similar absolute volumes (16-bit audio doesn't have enough room to accommodate such different volume levels). An application might use a minimum distance of 100 meters for the jet engine and 2 centimeters for the bee. With these settings, the jet engine would be at half volume when the listener was 200 meters away, but the bee would be at half volume when the listener was 4 centimeters away. This concept is shown in the following illustration:



Doppler Shift

DirectSound automatically creates Doppler shift effects for any buffer or listener that has a *velocity*. Effects are cumulative: if the listener and the sound source are both moving, the system automatically calculates the relationship between their velocities and adjusts the Doppler effect accordingly.

The velocity of a sound source or listener does not necessarily reflect the speed at which it is moving through space. Setting an object's velocity does not move it, nor does moving the object affect the velocity. Velocity is simply a vector used to calculate the Doppler shift. In order to have realistic Doppler shift effects in your application, you must calculate the velocity of any object that is moving and set the appropriate value for that sound source or listener. You are free to exaggerate or minimize this value in order to create special effects.

You can also globally increase or decrease Doppler shift effects by setting the *Doppler factor* for the listener.

Integration with Direct3D

The **IDirectSound3DBuffer** and **IDirectSound3DListener** interfaces are designed to work together with Direct3D®. The positioning information used by Direct3D to arrange objects in a virtual environment can also be used to arrange sound sources. The **D3DVECTOR** and **D3DVALUE** types that are familiar to Direct3D programmers are also used in the **IDirectSound3DBuffer** and **IDirectSound3DListener** interfaces. The same left-handed coordinate system used by Direct3D is employed by DirectSound. (For information about coordinate systems, see 3-D Coordinate Systems, in the Direct3D overview material.)

You can use the system callback mechanism of Direct3D to simplify the implementation of 3-D sound in your application. For example, you could use the **D3DRMFRAMEMOVECALLBACK** function to monitor the movement of a frame in an application and change the sonic environment only when a certain condition has been reached.

Mono and Stereo Sources

Stereo sound sources are not particularly useful in the 3-D sound environments of DirectSound, because DirectSound creates its own stereo output from a monaural input. If an application uses stereo sound buffers, the left and right values for each sample are averaged before the 3-D processing is applied.

Applications should supply monaural sound sources when using the 3-D capabilities of DirectSound. Although the system can convert a stereo source into mono, there is no reason to supply stereo, and the conversion step wastes time.

DirectSound 3-D Buffers

A 3-D sound buffer is created and managed like any other sound buffer, and all the methods of the **IDirectSoundBuffer** interface are available. However, in order to set 3-D parameters you need to obtain the **IDirectSound3DBuffer** interface for the buffer. This interface is supported only by sound buffers successfully created with the DSBCAPS_CTRL3D flag.

This section describes how your applications can manage buffers with the **IDirectSound3DBuffer** interface methods. The following topics are discussed:

- Obtaining the **IDirectSound3DBuffer** Interface
- Batch Parameters for **IDirectSound3DBuffer**
- Minimum and Maximum Distances
- Operation Mode
- Buffer Position and Velocity
- Cone Parameters

Obtaining the IDirectSound3DBuffer Interface

To obtain a pointer to an **IDirectSound3DBuffer** interface, you must first create a secondary 3-D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D flag in the **dwFlags** member of the **DSBUFFERDESC** structure parameter. Then, use the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DBuffer** interface for that buffer.

The following example calls the **QueryInterface** method with the C++ syntax:

```
// LPDIRECTSOUNDBUFFER lpDsbSecondary;
// The buffer has been created with DSBCAPS_CTRL3D.
LPDIRECTSOUND3DBUFFER lpDs3dBuffer;

HRESULT hr = lpDsbSecondary->QueryInterface(IID_IDirectSound3DBuffer,
                                            &lpDs3dBuffer);
if (SUCCEEDED(hr))
  {
  // Set 3-D parameters of this sound.
    .
    .
    .
  }
```

Note Pan control conflicts with 3-D processing. If both DSBCAPS_CTRL3D and DSBCAPS_CTRLPAN are specified when the buffer is created, DirectSound returns an error.

Batch Parameters for IDirectSound3DBuffer

Applications can retrieve or set a 3-D sound buffer's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DBuffer** interface method. However, applications often must set or retrieve all the values at once. You can do this with the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods.

Minimum and Maximum Distances

Applications can specify the distances at which 3-D sounds stop getting louder or quieter. For an overview of these values, see Distance Measurements.

The default minimum distance, defined in Dsound.h as DS3D_DEFAULTMINDISTANCE, is currently 1 distance unit (normally 1 meter). The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

An application sets and retrieves the minimum distance value by using the **IDirectSound3DBuffer::SetMinDistance** and **IDirectSound3DBuffer::GetMinDistance** methods. Similarly, it can set and retrieve the maximum distance value by using the **IDirectSound3DBuffer::SetMaxDistance** and **IDirectSound3DBuffer::GetMaxDistance** methods.

By default, distance values are expressed in meters. See Distance Factor.

Operation Mode

Sound buffers have three processing modes: normal, head-relative, and disabled. Normal processing is the default mode. In the head-relative mode, sound parameters (position, velocity, and orientation) are relative to the listener's parameters; in this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change. In the disabled mode, 3-D sound processing is disabled and the sound seems to originate from the center of the listener's head.

An application sets the mode for a 3-D sound buffer by using the **IDirectSound3DBuffer::SetMode** method. This method sets the operation mode based on the flag the application sets for the first parameter, *dwMode*.

Buffer Position and Velocity

An application can set and retrieve a sound source’s position in 3-D space by using the **IDirectSound3DBuffer::SetPosition** and **IDirectSound3DBuffer::GetPosition** methods.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a listener, you use the **IDirectSound3DBuffer::SetVelocity** and **IDirectSound3DBuffer::GetVelocity** methods. A buffer's position is not affected by its velocity. Velocity is measured in distance units per second - by default, meters per second.

See also:

- Doppler Shift
- Distance Measurements

Cone Parameters

An application sets or retrieves the angles that define sound cones by using the **IDirectSound3DBuffer::SetConeAngles** and **IDirectSound3DBuffer::GetConeAngles** methods. To set or retrieve the orientation of sound cones, an application can use the **IDirectSound3DBuffer::SetConeOrientation** and **IDirectSound3DBuffer::GetConeOrientation** methods.

By default, cone angles are 360 degrees, meaning the object projects sound at the same volume in all directions. A smaller value means that the object projects sound at a lower volume outside the defined angle. The outside cone angle must always be equal to or greater than the inside cone angle.

The outside cone volume represents the additional volume attenuation of the sound when the listener is outside the buffer's sound cone. This factor is expressed in hundredths of decibels. By default the outside volume is zero, meaning the sound cone will have no perceptible effect.

An application sets and retrieves the outside cone volume by using the **IDirectSound3DBuffer::SetConeOutsideVolume** and **IDirectSound3DBuffer::GetConeOutsideVolume** methods. Keep in mind that an audible outside cone volume is still subject to attenuation, due to distance from the sound source.

When the listener is within the sound cone, the normal buffer volume (returned by the **IDirectSoundBuffer::GetVolume** method) is used.

For a conceptual overview, see Sound Cones.

DirectSound 3-D Listeners

A 3-D listener represents the person who hears sounds generated by sound buffer objects in 3-D space. The **IDirectSound3DListener** interface controls the listener's position and apparent velocity in 3-D space. It also controls the environment parameters that affect the behavior of the DirectSound component, such as the amount of Doppler shifting and volume attenuation applied to sound sources far from the listener.

This section describes how your application can obtain a pointer to an **IDirectSound3DListener** interface and manage listener parameters by using interface methods. The following topics are discussed:

- Obtaining the **IDirectSound3DListener** Interface
- Batch Parameters for **IDirectSound3DListener**
- Deferred Settings
- Distance Factor
- Doppler Factor
- Listener Position and Velocity
- Listener Orientation
- Rolloff Factor

Obtaining the IDirectSound3DListener Interface

To obtain a pointer to an **IDirectSound3DListener** interface, you must first create a primary 3-D sound buffer. Do this by using the **IDirectSound::CreateSoundBuffer** method, specifying the DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER flags in the **dwFlags** member of the accompanying **DSBUFFERDESC** structure. Then, use the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DListener** interface for that buffer, as shown in the following example with C++ syntax:

```
// LPDIRECTSOUNDBUFFER lpDsbPrimary;
// The buffer has been created with DSBCAPS_CTRL3D.
LPDIRECTSOUND3DLISTENER lpDs3dListener;

HRESULT hr = lpDsbPrimary->QueryInterface(IID_IDirectSound3DListener,
                                          &lpDs3dListener);

if (SUCCEEDED(hr))
  {
  // Perform 3-D operations.
  .
  .
  .
  }
```

Batch Parameters for IDirectSound3DListener

Applications can retrieve or set a 3-D listener object's parameters individually or in batches. To set individual values, your application can use the applicable **IDirectSound3DListener** interface method. However, applications often must set or retrieve all the values that describe the listener at once. An application can perform these batch parameter manipulations in a single call by using the **IDirectSound3DListener::GetAllParameters** and **IDirectSound3DListener::SetAllParameters** methods.

Deferred Settings

Changes to 3-D sound buffer and listener settings such as position, velocity, and Doppler factor will cause the DirectSound mixer to remix its mix-ahead buffer, at the expense of CPU cycles. To minimize the performance impact of changing 3-D settings, use the DS3D_DEFERRED flag in the *dwApply* parameter of any of the **IDirectSound3DListener** or **IDirectSound3DBuffer** methods that change 3-D settings. Then call the **IDirectSound3DListener::CommitDeferredSettings** method to execute all of the deferred commands with a single remix of the mix-ahead buffer.

Note Any deferred settings are overwritten if your application calls the same setting method with the DS3D_IMMEDIATE flag before it calls **IDirectSound3DListener::CommitDeferredSettings**. For example, if you set the listener velocity to (1,2,3) with the deferred flag and then set the listener velocity to (4,5,6) with the immediate flag, the velocity will be (4,5,6). Then, if your application calls the **IDirectSound3DListener::CommitDeferredSettings** method, the velocity will still be (4,5,6).

Distance Factor

DirectSound uses meters as the default unit of distance measurements. If your application does not use meters, it can set a distance factor. For an overview, see Distance Measurements.

After you have set the distance factor for a listener, use your application's own distance units in calls to any methods that apply to that listener. Suppose, for example, that the basic unit of measurement in your application is the foot. You call the **IDirectSound3DListener::SetDistanceFactor** method, specifying 0.3048 as the *flDistanceFactor* parameter. (This value is the number of meters in a foot.) From then on, you continue using feet in parameters to method calls, and they are automatically converted to meters.

You can retrieve the current distance factor set for a listener with the **IDirectSound3DListener::GetDistanceFactor** method. The default value is DS3D_DEFAULTDISTANCEFACTOR, defined as 1.0, meaning that a distance unit corresponds to 1 meter. At the default value, a position vector of (3.0,7.2,-20.9) means that the object is 3.0 m to the right of, 7.2 m above, and 20.9 m behind the origin. If the distance factor is changed to 2.0, the same position vector means that the object is 6.0 m to the right of, 14.4 m above, and 41.8 m behind the origin.

Doppler Factor

DirectSound applies Doppler-shift effects to sounds, based on the relative velocity of the listener and the sound buffer. (For an overview, see Doppler Shift.) The Doppler shift can be ignored, exaggerated, or given the same effect as in the real world, depending on a variable called the Doppler factor.

The Doppler factor can range from DS3D_MINDOPPLERFACTOR to DS3D_MAXDOPPLERFACTOR, currently defined in Dsound.h as 0.0 and 10.0 respectively. A value of 0 means no Doppler shift is applied to a sound. Every other value represents a multiple of the real-world Doppler shift. In other words, a value of 1 (or DS3D_DEFAULTDOPPLERFACTOR) means the Doppler shift that would be experienced in the real world is applied to the sound; a value of 2 means two times the real-world Doppler shift; and so on.

The Doppler factor can be set and retrieved with the **IDirectSound3DListener::SetDopplerFactor** and **IDirectSound3DListener::GetDopplerFactor** methods.

Listener Position and Velocity

An application can set and retrieve a listener's position in 3-D space by using the **IDirectSound3DListener::SetPosition** and **IDirectSound3DListener::GetPosition** methods.

To set or retrieve the velocity value that DirectSound uses to calculate Doppler-shift effects for a listener, use the **IDirectSound3DListener::SetVelocity** and **IDirectSound3DListener::GetVelocity** methods. A listener's position is not affected by its velocity.

Listener Orientation

The listener's orientation plays a strong role in 3-D effects processing. DirectSound approximates sound cues to provide the illusion that a sound is generated at a particular point in space. For more information about these cues, see Perception of Sound Positions.

An application can set and retrieve the listener's orientation by using the **IDirectSound3DListener::SetOrientation** and **IDirectSound3DListener::GetOrientation** methods. By default, the front vector is (0,0,1.0), and the top vector is (0,1.0,0).

For an illustration of the front and top vectors, see Listeners.

Rolloff Factor

Rolloff is the amount of attenuation that is applied to sounds, based on the listener's distance from the sound source. DirectSound can ignore rolloff, exaggerate it, or give it the same effect as in the real world, depending on a variable called the rolloff factor.

The rolloff factor can range from DS3D_MINROLLOFFFACTOR to DS3D_MAXROLLOFFFACTOR, currently defined in Dsound.h as 0.0 and 10.0 respectively. A value of DS3D_MINROLLOFFFACTOR means no rolloff is applied to a sound. Every other value represents a multiple of the real-world rolloff. In other words, a value of 1 (DS3D_DEFAULTROLLOFFFACTOR) means the rolloff that would be experienced in the real world is applied to the sound; a value of 2 means two times the real-world rolloff, and so on.

You set and retrieve the rolloff factor by using the **IDirectSound3DListener::SetRolloffFactor** and **IDirectSound3DListener::GetRolloffFactor** methods.

DirectSoundCapture

DirectSoundCapture provides an interface for capturing digital audio data from an input source. To use it you must create an instance of the **IDirectSoundCapture** interface, then use its methods to create a single capture buffer. (The present version of DirectSound does not permit capturing and mixing from multiple devices at the same time.) The actual capturing is done with the methods of the buffer object.

This section covers the following topics:

Creating the DirectSoundCapture Object

You create the DirectSoundCapture object by calling the **DirectSoundCaptureCreate** function, which returns a pointer to an **IDirectSoundCapture** COM interface.

You can also use the **CoCreateInstance** function to create the object. The procedure is similar to that for the DirectSound object; see Creating the DirectSound Object. If you use **CoCreateInstance**, then the object is created for the default capture device selected by the user on the multimedia control panel.

If you want DirectSound and DirectSoundCapture objects to coexist, then you should create and initialize the DirectSound object before creating and initializing the DirectSoundCapture object. Some audio devices aren't configured for full duplex audio by default. If you have problems with creating and initializing both a DirectSound object and a DirectSoundCapture object, you should check your audio device to ensure that two DMA channels are enabled.

Capture Device Capabilities

To retrieve the capabilities of a capture device, call the **IDirectSoundCapture::GetCaps** method. The argument to this method is a **DSCCAPS** structure. As with other such structures, you have to initialize the **dwSize** member before passing it as an argument.

On return, the structure contains the number of channels the device supports as well as a combination of values for supported formats, equivalent to the values in the **WAVEINCAPS** structure used in the Win32 waveform audio functions. These are reproduced here for convenience.

Value	Meaning
WAVE_FORMAT_1M08	11.025 kHz, mono, 8-bit
WAVE_FORMAT_1M16	11.025 kHz, mono, 16-bit
WAVE_FORMAT_1S08	11.025 kHz, stereo, 8-bit
WAVE_FORMAT_1S16	11.025 kHz, stereo, 16-bit
WAVE_FORMAT_2M08	22.05 kHz, mono, 8-bit
WAVE_FORMAT_2M16	22.05 kHz, mono, 16-bit
WAVE_FORMAT_2S08	22.05 kHz, stereo, 8-bit
WAVE_FORMAT_2S16	22.05 kHz, stereo, 16-bit
WAVE_FORMAT_4M08	44.1 kHz, mono, 8-bit
WAVE_FORMAT_4M16	44.1 kHz, mono, 16-bit
WAVE_FORMAT_4S08	44.1 kHz, stereo, 8-bit
WAVE_FORMAT_4S16	44.1 kHz, stereo, 16-bit

Creating a Capture Buffer

Create a capture buffer by calling the **IDirectSoundCapture::CreateCaptureBuffer** method of the DirectSoundCapture object.

One of the parameters to the method is a **DSCBUFFERDESC** structure that describes the characteristics of the desired buffer. The last member of this structure is a **WAVEFORMATEX** structure, which must be initialized with the details of the desired wave format. See the reference for **WAVEFORMATEX** in the Win32 API section of the Platform SDK for information on the members of that structure.

The following example sets up a capture buffer that will hold about 1 second of data:

```
DSCBUFFERDESC              dscbd;
LPDIRECTSOUNDCAPTUREBUFFER pDSCB;
WAVEFORMATEX               wfx =
{
    // wFormatTag, nChannels, nSamplesPerSec, mAvgBytesPerSec,
    // nBlockAlign, wBitsPerSample, cbSize
    {WAVE_FORMAT_PCM, 2, 44100, 176400, 4, 16, 0},
};
dscbd.dwSize = sizeof(DSCBUFFERDESC);
dscbd.dwFlags = 0;
// We're going to capture one second's worth of audio
dscbd.dwBufferBytes = wfx.nAvgBytesPerSec;
dscbd.dwReserved = 0;
dscbd.lpwfxFormat = &wfx;

pDSCB = NULL;

// pDSC is the pointer to the DirectSoundCapture object
HRESULT hr = pDSC->CreateCaptureBuffer(&dscbd,
                                       &pDSCB, NULL);
```

Capture Buffer Information

Use the **IDirectSoundCaptureBuffer::GetCaps** method to retrieve the size of a capture buffer. Be sure to initialize the **dwSize** member of the **DSCBCAPS** structure before passing it as a parameter.

You can also retrieve information about the format of the data in the buffer, as set when the buffer was created. Call the **IDirectSoundCaptureBuffer::GetFormat** method, which returns the format information in a **WAVEFORMATEX** structure. See the reference for **WAVEFORMATEX** in the Win32 API section of the Platform SDK for information on the members of that structure.

Note that your application can allow for extra format information in the **WAVEFORMATEX** structure by first calling the **GetFormat** method with NULL as the *lpwfxFormat* parameter. In this case the **DWORD** pointed to by the *lpdwSizeWritten* parameter will receive the size of the structure needed to receive complete format information.

To find out what a capture buffer is currently doing, call the **IDirectSoundCaptureBuffer::GetStatus** method. This method fills a **DWORD** variable with a combination of flags that indicate whether the buffer is busy capturing, and if so, whether it is looping; that is, whether the DSCBSTART_LOOPING flag was set in the last call to **IDirectSoundCaptureBuffer::Start**.

Finally, the **IDirectSoundCaptureBuffer::GetCurrentPosition** method returns the current read and capture positions within the buffer. The read position is the end of the data that has been captured into the buffer at this point. The capture position is the end of the block of data that is currently being copied from the hardware. You can safely copy data from the buffer only up to the read position.

Capture Buffer Notification

You may want your application to be notified when the current read position reaches a certain point in the buffer, or when it reaches the end. The current read position is the point up to which it is safe to read data from the buffer. With the **IDirectSoundNotify::SetNotificationPositions** method you can set any number of points within the buffer where events are to be signaled.

First you have to obtain a pointer to the **IDirectSoundNotify** interface. You can do this with the capture buffer's **QueryInterface** method, as shown in the example under Play Buffer Notification.

Next create an event object with the Win32 **CreateEvent** function. You put the handle to this event in the **hEventNotify** member of a **DSBPOSITIONNOTIFY** structure, and in the **dwOffset** member of that structure you specify the offset within the buffer where you want the event to be signaled. Then you pass the address of the structure—or of an array of structures, if you want to set more than one notification position—to the **IDirectSoundNotify::SetNotificationPositions** method.

The following example sets up three notification positions. One event will be signaled when the read position nears the halfway point in the buffer, another will be signaled when it nears the end of the buffer, and the third will be signaled when capture stops.

```
#define cEvents  3

//  LPDIRECTSOUNDNOTIFY  lpDsNotify;
//    lpDsNotify was initialized with QueryInterface.
//  WAVEFORMATEX   wfx;
//    wfx was initialized when the buffer was created.
HANDLE             rghEvent[cEvents] = {0};
DSBPOSITIONNOTIFY  rgdsbpn[cEvents];
HRESULT            hr;
int                i;

// create the events
for (i = 0; i < cEvents; ++i)
{
    rghEvent[i] = CreateEvent(NULL, TRUE, FALSE, NULL);
    if (NULL == rghEvent[i])
    {
        hr = GetLastError();
        goto Error;
    }
}

// Set notification positions.
// Notify us when read position is halfway through the buffer,
// assuming buffer holds one second of audio.
rgdsbpn[0].dwOffset = (wfx.nAvgBytesPerSec/2) -1;
rgdsbpn[0].hEventNotify = rghEvent[0];
// Notify us when capture is at the end of the buffer.
rgdsbpn[1].dwOffset = wfx.nAvgBytesPerSec - 1;
rgdsbpn[1].hEventNotify = rghEvent[1];
rgdsbpn[2].dwOffset = DSBPN_OFFSETSTOP;
rgdsbpn[2].hEventNotify = rghEvent[2];

hr = lpDsNotify->SetNotificationPositions(cEvents, rgdsbpn);
```

Capturing Sounds

Capturing a sound consists of the following steps:

1. Start the buffer by calling the **IDirectSoundCaptureBuffer::Start** method. Audio data from the input device begins filling the buffer from the beginning.
2. Wait until the desired amount of data is available. See Capture Buffer Notification for one method of determining when the capture position reaches a certain point.
3. When sufficient data is available, lock a portion of the capture buffer by calling the **IDirectSoundCaptureBuffer::Lock** method.

   To make sure you are not attempting to lock a portion of memory that is about to be used for capture, you should first obtain the current read position by calling **IDirectSoundCaptureBuffer::GetCurrentPosition**. For an explanation of the read position, see Capture Buffer Information.

   As parameters to the **Lock** method, you pass the size and offset of the block of memory you want to read. The method returns a pointer to the address where the memory block begins, and the size of the block. If the block wraps around from the end of the buffer to the beginning, two pointers are returned, one for each section of the block. The second pointer is NULL if the locked portion of the buffer does not wrap around.
4. Copy the data from the buffer, using the addresses and block sizes returned by the **Lock** method.
5. Unlock the buffer with the **IDirectSoundCaptureBuffer::Unlock** method.
6. Repeat steps 2 to 5 until all the data is captured. Then call the **IDirectSoundCaptureBuffer::Stop** method.

Normally the buffer stops capturing automatically when the capture position reaches the end of the buffer. However, if the DSCBSTART_LOOPING flag was set in the *dwFlags* parameter to the **IDirectSoundCaptureBuffer::Start** method, the capture will continue until the application calls the **IDirectSoundCaptureBuffer::Stop** method, at which point the capture position is moved to the beginning of the buffer.

DirectSound Property Sets

Through the **IKsPropertySet** interface, DirectSound is able to support extended services offered by sound cards and their associated drivers.

Properties are arranged in sets. A **GUID** identifies a set, and a **ULONG** identifies a particular property within the set. For example, a hardware vendor might design a card capable of reverberation effects and define a property set **DSPROPSETID_ReverbProperties** containing properties such as **DSPROPERTY_REVERBPROPERTIES_HALL** and **DSPROPERTY_REVERBPROPERTIES_STADIUM**.

Typically, the property identifiers are defined using a C language enumeration starting at ordinal 0.

Individual properties may also have associated parameters. The **IKsPropertySet** interface specification intentionally leaves these parameters undefined, allowing the designer of the property set to use them in a way most beneficial to the properties within the set being designed. The precise meaning of the parameters is defined with the definition of the properties.

To make use of extended properties on sound cards, you must first determine whether the driver supports the **IKsPropertySet** interface, and obtain a pointer to the interface if it is supported. You can do this by calling the **QueryInterface** method of an existing interface on a DirectSound3DBuffer object.

```
HRESULT hr = lpDirectSound3DBuffer->QueryInterface(
                                IID_IKsPropertySet,
                                (void**)&lpKsPropertySet))
```

In the example, *lpDirectSound3DBuffer* is a pointer to the buffer's interface and *lpKsPropertySet* receives the address of the **IKsPropertySet** interface if one is found. *IID_IKsPropertySet* is a **GUID** defined in Dsound.h.

The call will succeed only if the buffer is hardware-accelerated and the underlying driver supports property sets. If it does succeed, you can now look for a particular property using the **IKsPropertySet::QuerySupport** method. The value of the *PropertySetId* parameter is a **GUID** defined by the hardware vendor.

Once you've determined that support for a particular property exists, you can change the state of the property by using the **IKsPropertySet::Set** method and determine its present state by using the **IKsPropertySet::Get** method. The state of the property is set or returned in the *pPropertyData* parameter.

Additional property parameters may also be passed to the object in a structure pointed to by the *pPropertyParams* parameter to the **IKsPropertySet::Set** method. The exact way in which this parameter is to be used is defined in the hardware vendor's specifications for the property set, but typically it would be used to define the instance of the property set. In practice, the *pPropertyParams* parameter is rarely used.

Let's take a somewhat whimsical example. Suppose a sound card has the ability to play famous arias in the voices of several tenors. The driver developer creates a property set, **DSPROPSETID_Aria**, containing properties like **DSPROPERTY_ARIA_VESTI_LA_GIUBBA** and **DSPROPERTY_ARIA_CHE_GELIDA_MANINA**. The property set applies to all of the tenors, and the driver developer has specified that *pPropertyParams* defines the tenor instance. Now you, the application developer, want to make Caruso sing the great aria from *Pagliacci*.

```
DWORD WhichTenor = CARUSO;
BOOL  StartOrStop = START;

HRESULT hr = lpKsPropertySet->Set(
                    DSPROPSETID_Aria,
                    KSPROPERTY_ARIA_VESTI_LA_GIUBBA,
                    &WhichTenor,
```

```
sizeof(WhichTenor),
&StartOrStop),
sizeof(StartOrStop);
```

Optimizing DirectSound Performance

This section offers some miscellaneous tips for improving the performance of DirectSound. The following topics are covered:

- Matching Buffer Formats
- Reducing DMA Overhead
- Playing the Primary Buffer Continuously
- Using Hardware Mixing
- Minimizing Control Changes
- CPU Considerations for 3-D Buffers

Matching Buffer Formats

The DirectSound mixer converts the data from each secondary buffer into the format of the primary buffer. This conversion is done on the fly as data is mixed into the primary buffer, and costs CPU cycles. You can eliminate this overhead by ensuring that your secondary buffers (that is, wave files) and primary buffer have the same format.

Because of the way DirectSound does format conversion, you only need to match the sample rate and number of channels. It doesn't matter if there is a difference in sample size (8-bit or 16-bit).

Reducing DMA Overhead

Most of today's sound cards are ISA-bus cards that use DMA (direct memory access) to move sound data from system memory to local buffers. This DMA activity directly affects CPU performance when the processor is forced to wait for a DMA transfer to end before it can access memory. This performance hit is unavoidable on ISA sound cards but is not a problem with the newer PCI cards.

DMA overhead could be the biggest single factor affecting the performance of DirectSound. Fortunately, this factor is easy to control when you're tweaking performance.

The impact of DMA overhead is directly related to the data rate of the primary buffer. Experiment with reducing the data rate requirement by changing the format of the primary buffer. For more information on how to do this, see Access to the Primary Buffer and **IDirectSoundBuffer::SetFormat**.

Playing the Primary Buffer Continuously

When there are no sounds playing, DirectSound stops the mixer engine and halts DMA (direct memory access) activity. If your application has frequent short intervals of silence, the overhead of starting and stopping the mixer each time a sound is played may be worse than the DMA overhead if you kept the mixer active. Also, some sound hardware or drivers may produce unwanted audible artifacts from frequent starting and stopping of playback. If your application is playing audio almost continuously with only short breaks of silence, you can force the mixer engine to remain active by calling the **IDirectSoundBuffer::Play** method for the primary buffer. The mixer will continue to run silently.

To resume the default behavior of stopping the mixer engine when there are no sounds playing, call the **IDirectSoundBuffer::Stop** method for the primary buffer.

For more information, see Access to the Primary Buffer

Using Hardware Mixing

Most sound cards support some level of hardware mixing if there is a DirectSound driver for the card. The following tips will allow you to make the most of hardware mixing:

- Use static buffers for sounds that you want to be mixed in hardware. DirectSound will attempt to use hardware mixing on static buffers.
- Create sound buffers first for the sounds you use the most. There is a limit to the number of buffers that can be mixed by hardware.
- At run time, use the **IDirectSound::GetCaps** method to determine what formats are supported by the sound-accelerator hardware and use only those formats if possible.
- To create a static buffer, specify the DSBCAPS_STATIC flag in the **dwFlags** member of the **DSBUFFERDESC** structure when you create a secondary buffer. You can also specify the DSBCAPS_LOCHARDWARE flag to force hardware mixing for a buffer, however, if you do this and resources for hardware mixing are not available, the **IDirectSound::CreateSoundBuffer** method will fail.

Minimizing Control Changes

Performance is affected when you change the pan, volume, or frequency on a secondary buffer. To prevent interruptions in sound output, the DirectSound mixer must mix ahead from 20 to 100 or more milliseconds. Whenever you make a control change, the mixer has to flush its mix-ahead buffer and remix with the changed sound.

It's a good idea to minimize the number of control changes you send, especially if you're sending them in streams or in bursts. Try reducing the frequency of calls to routines that use the **IDirectSoundBuffer**::**SetVolume**, **IDirectSoundBuffer::SetPan**, and **IDirectSoundBuffer::SetFrequency** methods. For example, if you have a routine that moves a sound from the left to the right speaker in synchronization with animation frames, try calling the **SetPan** method once instead of twice per frame.

Note 3-D control changes (orientation, position, velocity, Doppler factor, and so on) also cause the DirectSound mixer to remix its mix-ahead buffer. However, you can group a number of 3-D control changes together and cause only a single remix. See Deferred Settings.

CPU Considerations for 3-D Buffers

Software-emulated 3-D buffers are computationally expensive. For example, each buffer can consume about 6 percent of the processing time of a Pentium 90. You should take this into consideration when deciding when and how to use 3-D buffers in your applications.

Use as few 3-D sounds as you can, and don't use 3-D on sounds that won't really benefit from the effect. Design your application so that it's easy to enable and disable 3-D effects on each sound. You can call the **IDirectSound3DBuffer::SetMode** method with the DS3DMODE_DISABLE flag to disable 3-D processing on any 3-D sound buffer.

DirectSound provides a means for hardware manufacturers to provide acceleration of 3-D audio buffers. On these audio cards the host CPU consumption will not be a consideration.

DirectSound Reference

This section contains reference information for the API elements that DirectSound provides. Reference material is divided into the following categories.

- Interfaces
- Functions
- Callback Function
- Structures
- Return Values

Interfaces

This section contains references for methods of the following DirectSound interfaces:

- **IDirectSound**
- **IDirectSound3DBuffer**
- **IDirectSound3DListener**
- **IDirectSoundCapture**
- **IDirectSoundCaptureBuffer**
- **IDirectSoundNotify**
- **IKsPropertySet**

IDirectSound

Applications use the methods of the **IDirectSound** interface to create DirectSound objects and set up the environment. This section is a reference to the methods of this interface.

The interface is obtained by using the **DirectSoundCreate** function.

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

Allocating memory	**Compact** **Initialize**
Creating buffers	**CreateSoundBuffer** **DuplicateSoundBuffer** **SetCooperativeLevel**
Device capabilities	**GetCaps**
Speaker configuration	**GetSpeakerConfig** **SetSpeakerConfig**

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

AddRef
QueryInterface
Release

The **LPDIRECTSOUND** type is defined as a pointer to the **IDirectSound** interface:

```
typedef struct IDirectSound    *LPDIRECTSOUND;
```

IDirectSound::Compact

The **IDirectSound::Compact** method moves the unused portions of on-board sound memory, if any, to a contiguous block so that the largest portion of free memory will be available.

```
HRESULT Compact();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED
DSERR_UNINITIALIZED

Remarks

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

The **IDirectSound::CreateSoundBuffer** method creates a DirectSoundBuffer object to hold a sequence of audio samples.

```
HRESULT CreateSoundBuffer(
  LPCDSBUFFERDESC lpcDSBufferDesc,
  LPLPDIRECTSOUNDBUFFER lplpDirectSoundBuffer,
  IUnknown FAR * pUnkOuter
);
```

Parameters

lpcDSBufferDesc
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.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_BADFORMAT
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

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 on-board memory, if available, to take advantage of hardware mixing. To force the buffer to use either hardware or software mixing, use the DSBCAPS_LOCHARDWARE or DSBCAPS_LOCSOFTWARE flag.

See Also

DSBUFFERDESC, **IDirectSound::DuplicateSoundBuffer**, **IDirectSound::SetCooperativeLevel**, *IDirectSoundBuffer*, **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::SetFormat**, **IDirectSoundBuffer::SetFrequency**

IDirectSound::DuplicateSoundBuffer

The **IDirectSound::DuplicateSoundBuffer** method creates a new DirectSoundBuffer object that uses the same buffer memory as the original object.

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

Parameters

lpDsbOriginal
Address of the DirectSoundBuffer object to be duplicated.

lplpDsbDuplicate
Address of a pointer to the new DirectSoundBuffer object.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

Remarks

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 because the buffer memory is shared.

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

Applications cannot assume that an attempt to duplicate a sound buffer will always succeed. In particular, DirectSound will not create a software duplicate of a hardware buffer.

See Also

IDirectSound::CreateSoundBuffer

IDirectSound::GetCaps

The **IDirectSound::GetCaps** method retrieves the capabilities of the hardware device that is represented by the DirectSound object.

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

Parameters

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

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

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 on-board sound memory. You can use this information to fine-tune performance and optimize resource allocation.

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

See Also

DirectSoundCreate, **DSCAPS**

IDirectSound::GetSpeakerConfig

The **IDirectSound::GetSpeakerConfig** method retrieves the speaker configuration specified for this DirectSound object.

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

Parameters

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 played through headphones.

DSSPEAKER_MONO

The audio is played through a single speaker.

DSSPEAKER_QUAD

The audio is played through quadraphonic speakers.

DSSPEAKER_STEREO

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

DSSPEAKER_SURROUND

The audio is played through surround speakers.

DSSPEAKER_STEREO may be combined with one of the following values:

DSSPEAKER_GEOMETRY_WIDE

The speakers are directed over an arc of 20 degrees

DSSPEAKER_GEOMETRY_NARROW

The speakers are directed over an arc of 10 degrees

DSSPEAKER_GEOMETRY_MIN

The speakers are directed over an arc of 5 degrees

DSSPEAKER_GEOMETRY_MAX

The speakers are directed over an arc of 180 degrees

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

The value returned at *lpdwSpeakerConfig* may be a packed **DWORD** containing both configuration and geometry information. Use the **DSSPEAKER_CONFIG** and **DSSPEAKER_GEOMETRY** macros to

unpack the **DWORD**, as in the following example:

```
if (DSSPEAKER_CONFIG(dwSpeakerConfig) == DSSPEAKER_STEREO)
    {
    if (DSSPEAKER_GEOMETRY(dwSpeakerConfig) ==
                      DSSPEAKER_GEOMETRY_WIDE)
       {...}
    }
```

See Also

IDirectSound::SetSpeakerConfig

IDirectSound::Initialize

The **IDirectSound::Initialize** method initializes the DirectSound object that was created by using the **CoCreateInstance** function.

```
HRESULT Initialize(
  LPGUID lpGuid
);
```

Parameters

lpGuid

Address of the globally unique identifier (GUID) specifying the sound driver for this DirectSound object to bind to. Pass NULL to select the primary sound driver.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALREADYINITIALIZED
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_NODRIVER

Remarks

This method is provided for compliance with the Component Object Model (COM) protocol. If the **DirectSoundCreate** function was used to create the DirectSound object, this method returns DSERR_ALREADYINITIALIZED. If **IDirectSound::Initialize** is not called when using **CoCreateInstance** to create the DirectSound object, any method called afterward returns DSERR_UNINITIALIZED.

See Also

DirectSoundCreate

IDirectSound::SetCooperativeLevel

The **IDirectSound::SetCooperativeLevel** method sets the cooperative level of the application for this sound device.

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

Parameters

hwnd

Window handle to 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 (sounds from applications with the DSBCAPS_GLOBALFOCUS flag set will be muted). 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. (Note that DirectSound will always restore the wave format no matter what priority level is set.)

DSSCL_NORMAL

Sets the application to a fully cooperative status. Most applications should use this level, because it has the smoothest multitasking 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 sound buffers. No secondary sound buffers can be played. This level cannot be set if the DirectSound driver is being emulated for the device; that is, if the **IDirectSound::GetCaps** method returns the DSCAPS_EMULDRIVER flag in the **DSCAPS** structure.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_UNINITIALIZED
DSERR_UNSUPPORTED

Remarks

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. For additional information, see Cooperative Levels.

See Also

IDirectSound::Compact, **IDirectSoundBuffer::GetFormat**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::Restore**, **IDirectSoundBuffer::SetFormat**

IDirectSound::SetSpeakerConfig

The **IDirectSound::SetSpeakerConfig** method specifies the speaker configuration of the DirectSound object.

```
HRESULT SetSpeakerConfig(
  DWORD dwSpeakerConfig
);
```

Parameters

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.

DSSPEAKER_STEREO may be combined with one of the following values:

DSSPEAKER_GEOMETRY_WIDE

The speakers are directed over an arc of 20 degrees

DSSPEAKER_GEOMETRY_NARROW

The speakers are directed over an arc of 10 degrees

DSSPEAKER_GEOMETRY_MIN

The speakers are directed over an arc of 5 degrees

DSSPEAKER_GEOMETRY_MAX

The speakers are directed over an arc of 180 degrees

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNINITIALIZED

Remarks

If a geometry value is to be used, it must be packed in a **DWORD** along with the DSSPEAKER_STEREO flag. This can be done by using the **DSSPEAKER_COMBINED** macro, as in

the following C++ example:

```
lpds->SetSpeakerConfig(DSSPEAKER_COMBINED(
    DSSPEAKER_STEREO, DSSPEAKER_GEOMETRY_WIDE));
```

See Also

IDirectSound::GetSpeakerConfig

IDirectSound3DBuffer

Applications use the methods of the **IDirectSound3DBuffer** interface to retrieve and set parameters that describe the position, orientation, and environment of a sound buffer in 3-D space. This section is a reference to the methods of this interface. For a conceptual overview, see DirectSound 3-D Buffers.

The IDirectSound3DBuffer is obtaining by using the **IDirectSoundBuffer::QueryInterface** method. For more information, see Obtaining the IDirectSound3DBuffer Interface.

The methods of the **IDirectSound3DBuffer** interface can be organized into the following groups:

Batch parameter manipulation	**GetAllParameters** **SetAllParameters**
Distance	**GetMaxDistance** **GetMinDistance** **SetMaxDistance** **SetMinDistance**
Operation mode	**GetMode** **SetMode**
Position	**GetPosition** **SetPosition**
Sound projection cones	**GetConeAngles** **GetConeOrientation** **GetConeOutsideVolume** **SetConeAngles** **SetConeOrientation** **SetConeOutsideVolume**
Velocity	**GetVelocity** **SetVelocity**

The **IDirectSound3DBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUND3DBUFFER** type is defined as a pointer to the **IDirectSound3DBuffer** interface:

```
typedef struct IDirectSound3DBuffer    *LPDIRECTSOUND3DBUFFER;
```

IDirectSound3DBuffer::GetAllParameters

The **IDirectSound3DBuffer::GetAllParameters** method retrieves information that describes the 3-D characteristics of a sound buffer at a given point in time.

```
HRESULT GetAllParameters(
  LPDS3DBUFFER lpDs3dBuffer
);
```

Parameters

lpDs3dBuffer

Address of a **DS3DBUFFER** structure that will contain the information describing the 3-D characteristics of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::GetConeAngles

The **IDirectSound3DBuffer::GetConeAngles** method retrieves the inside and outside angles of the sound projection cone for this sound buffer.

```
HRESULT GetConeAngles(
  LPDWORD lpdwInsideConeAngle,
  LPDWORD lpdwOutsideConeAngle
);
```

Parameters

lpdwInsideConeAngle and *lpdwOutsideConeAngle*
Addresses of variables that will contain the inside and outside angles of the sound projection cone, in degrees.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The minimum, maximum, and default cone angles are defined in Dsound.h as DS3D_MINCONEANGLE, DS3D_MAXCONEANGLE, and DS3D_DEFAULTCONEANGLE.

IDirectSound3DBuffer::GetConeOrientation

The **IDirectSound3DBuffer::GetConeOrientation** method retrieves the orientation of the sound projection cone for this sound buffer.

```
HRESULT GetConeOrientation(
  LPD3DVECTOR lpvOrientation
);
```

Parameters

lpvOrientation

Address of a **D3DVECTOR** structure that will contain the current orientation of the sound projection cone. The vector information represents the center of the sound cone.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

This method has no effect unless the cone angle and cone volume factor have also been set. The default value is (0,0,1).

The values returned are not necessarily the same as those set by using the **IDirectSound3DBuffer::SetConeOrientation** method. DirectSound adjusts orientation vectors so that they are at right angles and have a magnitude of <= 1.0.

See Also

IDirectSound3DBuffer::SetConeOrientation,

IDirectSound3DBuffer::SetConeAngles, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::GetConeOutsideVolume

The **IDirectSound3DBuffer::GetConeOutsideVolume** method retrieves the current cone outside volume for this sound buffer.

```
HRESULT GetConeOutsideVolume(
  LPLONG lplConeOutsideVolume
);
```

Parameters

lplConeOutsideVolume
Address of a variable that will contain the current cone outside volume for this buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Volume levels are represented by attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). The default value is DS3D_DEFAULTCONEOUTSIDEVOLUME (no attenuation). These values are defined in Dsound.h. Currently DirectSound does not support amplification.

For additional information about the concept of outside volume, see Sound Cones.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::GetMaxDistance

The **IDirectSound3DBuffer::GetMaxDistance** method retrieves the current maximum distance for this sound buffer.

```
HRESULT GetMaxDistance(
  LPD3DVALUE lpflMaxDistance
);
```

Parameters

lpflMaxDistance
Address of a variable that will contain the current maximum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

See Also

IDirectSound3DBuffer::GetMinDistance, **IDirectSound3DBuffer::SetMaxDistance**

IDirectSound3DBuffer::GetMinDistance

The **IDirectSound3DBuffer::GetMinDistance** method retrieves the current minimum distance for this sound buffer.

```
HRESULT GetMinDistance(
  LPD3DVALUE lpflMinDistance
);
```

Parameters

lpflMinDistance
Address of a variable that will contain the current minimum distance setting.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

By default, the minimum distance value is DS3D_DEFAULTMINDISTANCE, currently defined as 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 m per unit).

See Also

IDirectSound3DBuffer::SetMinDistance, **IDirectSound3DBuffer::GetMaxDistance**

IDirectSound3DBuffer::GetMode

The **IDirectSound3DBuffer::GetMode** method retrieves the current operation mode for 3-D sound processing.

```
HRESULT GetMode(
  LPDWORD lpdwMode
);
```

Parameters

lpdwMode

Address of a variable that will contain the current mode setting. This value will be one of the following:

DS3DMODE_DISABLE

Processing of 3-D sound is disabled. The sound seems to originate from the center of the listener's head.

DS3DMODE_HEADRELATIVE

Sound parameters (position, velocity, and orientation) are relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

DS3DMODE_NORMAL

Normal processing. This is the default mode.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::GetPosition

The **IDirectSound3DBuffer::GetPosition** method retrieves the sound buffer's current position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetPosition(
  LPD3DVECTOR lpvPosition
);
```

Parameters

lpvPosition
Address of a **D3DVECTOR** structure that will contain the current position of the sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::GetVelocity

The **IDirectSound3DBuffer::GetVelocity** method retrieves the current velocity for this sound buffer. Velocity is measured in units per second. The default unit is one meter, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetVelocity(
  LPD3DVECTOR lpvVelocity
);
```

Parameters

lpvVelocity
Address of a **D3DVECTOR** structure that will contain the sound buffer's current velocity.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used for Doppler effects only. It does not actually move the buffer. For additional information, see Doppler Shift.

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

See Also

IDirectSound3DBuffer::SetPosition, **IDirectSound3DBuffer::SetVelocity**

IDirectSound3DBuffer::SetAllParameters

The **IDirectSound3DBuffer::SetAllParameters** method sets all 3-D sound buffer parameters from a given **DS3DBUFFER** structure that describes all aspects of the sound buffer at a moment in time.

```
HRESULT SetAllParameters(
  LPCDS3DBUFFER lpcDs3dBuffer,
  DWORD dwApply
);
```

Parameters

lpcDs3dBuffer

Address of a **DS3DBUFFER** structure containing the information that describes the 3-D characteristics of the sound buffer.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::SetConeAngles

The **IDirectSound3DBuffer::SetConeAngles** method sets the inside and outside angles of the sound projection cone for this sound buffer.

```
HRESULT SetConeAngles(
  DWORD dwInsideConeAngle,
  DWORD dwOutsideConeAngle,
  DWORD dwApply
);
```

Parameters

dwInsideConeAngle and *dwOutsideConeAngle*

Inside and outside angles of the sound projection cone, in degrees.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The minimum, maximum, and default cone angles are defined in Dsound.h as DS3D_MINCONEANGLE, DS3D_MAXCONEANGLE, and DS3D_DEFAULTCONEANGLE. Each angle must be in the range of 0 degrees (no cone) to 360 degrees (the full sphere). The default value is 360.

See Also

IDirectSound3DBuffer::GetConeOutsideVolume, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::SetConeOrientation

The **IDirectSound3DBuffer::SetConeOrientation** method sets the orientation of the sound projection cone for this sound buffer. This method has no effect unless the cone angle and cone volume factor have also been set.

```
HRESULT SetConeOrientation(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
Values whose types are **D3DVALUE** and that represent the coordinates of the new sound cone orientation vector.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The vector information in the *lpvOrientation* parameter of the **IDirectSound3DBuffer::GetConeOrientation** method represents the center of the sound cone. The default value is (0,0,1).

See Also

IDirectSound3DBuffer::SetConeAngles, **IDirectSound3DBuffer::SetConeOutsideVolume**

IDirectSound3DBuffer::SetConeOutsideVolume

The **IDirectSound3DBuffer::SetConeOutsideVolume** method sets the current cone outside volume for this sound buffer.

```
HRESULT SetConeOutsideVolume(
  LONG lConeOutsideVolume,
  DWORD dwApply
);
```

Parameters

lConeOutsideVolume

Cone outside volume for this sound buffer, in hundredths of decibels. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are defined in Dsound.h.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Volume levels are represented by attenuation. Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). The default value is DS3D_DEFAULTCONEOUTSIDEVOLUME (no attenuation). These values are defined in Dsound.h. Currently DirectSound does not support amplification.

For information about the concept of cone outside volume, see Sound Cones.

See Also

IDirectSoundBuffer::SetVolume

IDirectSound3DBuffer::SetMaxDistance

The **IDirectSound3DBuffer::SetMaxDistance** method sets the current maximum distance value.

```
HRESULT SetMaxDistance(
  D3DVALUE flMaxDistance,
  DWORD dwApply
);
```

Parameters

flMaxDistance

New maximum distance value.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The default maximum distance, defined as DS3D_DEFAULTMAXDISTANCE, is effectively infinite.

See Also

IDirectSound3DBuffer::GetMaxDistance, **IDirectSound3DBuffer::SetMinDistance**

IDirectSound3DBuffer::SetMinDistance

The **IDirectSound3DBuffer::SetMinDistance** method sets the current minimum distance value.

```
HRESULT SetMinDistance(
  D3DVALUE flMinDistance,
  DWORD dwApply
);
```

Parameters

flMinDistance

New minimum distance value.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

By default, the minimum distance value is DS3D_DEFAULTMINDISTANCE, currently defined as 1.0 (corresponding to 1.0 meter at the default distance factor of 1.0 m per unit).

See Also

IDirectSound3DBuffer::SetMaxDistance

IDirectSound3DBuffer::SetMode

The **IDirectSound3DBuffer::SetMode** method sets the operation mode for 3-D sound processing.

```
HRESULT SetMode(
  DWORD dwMode,
  DWORD dwApply
);
```

Parameters

dwMode

Flag specifying the 3-D sound processing mode to be set.

DS3DMODE_DISABLE

Processing of 3-D sound is disabled. The sound seems to originate from the center of the listener's head.

DS3DMODE_HEADRELATIVE

Sound parameters (position, velocity, and orientation) are relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

DS3DMODE_NORMAL

Normal processing. This is the default mode.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::SetPosition

The **IDirectSound3DBuffer::SetPosition** method sets the sound buffer's current position, in distance units. By default, distance units are meters, but the units can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT SetPosition(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*

Values whose types are **D3DVALUE** and that represent the coordinates of the new position vector.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSound3DBuffer::SetVelocity

The **IDirectSound3DBuffer::SetVelocity** method sets the sound buffer's current velocity.

```
HRESULT SetVelocity(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*

Values whose types are **D3DVALUE** and that represent the coordinates of the new velocity vector.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used for Doppler effects only. It does not actually move the buffer. For additional information, see Doppler Shift.

The default unit of measurement is meters per second, but this can be changed by using the **IDirectSound3DListener::SetDistanceFactor** method.

See Also

IDirectSound3DBuffer::SetPosition, **IDirectSound3DBuffer::GetVelocity**

IDirectSound3DListener

Applications use the methods of the **IDirectSound3DListener** interface to retrieve and set parameters that describe a listener's position, orientation, and listening environment in 3-D space. This section is a reference to the methods of this interface. For a conceptual overview, see DirectSound 3-D Listeners.

The interface is obtained by using the **IDirectSoundBuffer::QueryInterface** method. For more information, see Obtaining the IDirectSound3DListener Interface.

The methods of the **IDirectSound3DListener** interface can be organized into the following groups:

Batch parameter manipulation	**GetAllParameters** **SetAllParameters**
Deferred settings	**CommitDeferredSettings**
Distance factor	**GetDistanceFactor** **SetDistanceFactor**
Doppler factor	**GetDopplerFactor** **SetDopplerFactor**
Orientation	**GetOrientation** **SetOrientation**
Position	**GetPosition** **SetPosition**
Rolloff factor	**GetRolloffFactor** **SetRolloffFactor**
Velocity	**GetVelocity** **SetVelocity**

The **IDirectSound3DListener** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUND3DLISTENER** type is defined as a pointer to the **IDirectSound3DListener** interface:

```
typedef struct IDirectSound3DListener    *LPDIRECTSOUND3DLISTENER;
```

IDirectSound3DListener::CommitDeferredSettings

The **IDirectSound3DListener::CommitDeferredSetting** method commits any deferred settings made since the last call to this method.

```
HRESULT CommitDeferredSettings();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about using deferred settings to maximize efficiency, see Deferred Settings.

IDirectSound3DListener::GetAllParameters

The **IDirectSound3DListener::GetAllParameters** method retrieves information that describes the current state of the 3-D world and listener.

```
HRESULT GetAllParameters(
  LPDS3DLISTENER lpListener
);
```

Parameters

lpListener

Address of a **DS3DLISTENER** structure that will contain the current state of the 3-D world and listener.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

See Also

IDirectSound3DListener::SetAllParameters

IDirectSound3DListener::GetDistanceFactor

The **IDirectSound3DListener::GetDistanceFactor** method retrieves the current distance factor.

```
HRESULT GetDistanceFactor(
  LPD3DVALUE lpflDistanceFactor
);
```

Parameters

lpflDistanceFactor

Address of a variable whose type is **D3DVALUE** and that will contain the current distance factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about distance factors, see Distance Factor.

See Also

IDirectSound3DListener::SetDistanceFactor

IDirectSound3DListener::GetDopplerFactor

The **IDirectSound3DListener::GetDopplerFactor** method retrieves the current Doppler effect factor.

```
HRESULT GetDopplerFactor(
  LPD3DVALUE lpflDopplerFactor
);
```

Parameters

lpflDopplerFactor

Address of a variable whose type is **D3DVALUE** and that will contain the current Doppler factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The Doppler factor has a range of DS3D_MINDOPPLERFACTOR (no Doppler effects) to DS3D_MAXDOPPLERFACTOR (as currently defined, 10 times the Doppler effects found in the real world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information, see Doppler Factor.

See Also

IDirectSound3DListener::SetDopplerFactor

IDirectSound3DListener::GetOrientation

The **IDirectSound3DListener::GetOrientation** method retrieves the listener's current orientation in vectors: a front vector and a top vector.

```
HRESULT GetOrientation(
  LPD3DVECTOR lpvOrientFront,
  LPD3DVECTOR lpvOrientTop
);
```

Parameters

lpvOrientFront
Address of a **D3DVECTOR** structure that will contain the listener's front orientation vector.

lpvOrientTop
Address of a **D3DVECTOR** structure that will contain the listener's top orientation vector.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The front vector points in the direction of the listener's nose, and the top vector points out the top of the listener's head. By default, the front vector is (0,0,1.0) and the top vector is (0,1.0,0).

The values returned are not necessarily the same as those set by using the **IDirectSound3DListener::SetOrientation** method. DirectSound adjusts orientation vectors so that they are at right angles and have a magnitude of <= 1.0.

See Also

IDirectSound3DListener::SetOrientation

IDirectSound3DListener::GetPosition

The **IDirectSound3DListener::GetPosition** method retrieves the listener's current position in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT GetPosition(
  LPD3DVECTOR lpvPosition
);
```

Parameters

lpvPosition
Address of a **D3DVECTOR** structure that will contain the listener's position vector.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

See Also

IDirectSound3DListener::SetPosition

IDirectSound3DListener::GetRolloffFactor

The **IDirectSound3DListener::GetRolloffFactor** method retrieves the current rolloff factor.

```
HRESULT GetRolloffFactor(
  LPD3DVALUE lpflRolloffFactor
);
```

Parameters

lpflRolloffFactor

Address of a variable whose type is **D3DVALUE** and that will contain the current rolloff factor value.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information, see Rolloff Factor.

See Also

IDirectSound3DListener::SetRolloffFactor

IDirectSound3DListener::GetVelocity

The **IDirectSound3DListener::GetVelocity** method retrieves the listener's current velocity.

```
HRESULT GetVelocity(
  LPD3DVECTOR lpvVelocity
);
```

Parameters

lpvVelocity
Address of a **D3DVECTOR** structure that will contain the listener's current velocity.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used only for Doppler effects. It does not actually move the listener. To change the listener's position, use the **IDirectSound3DListener::SetPosition** method. The default velocity is (0,0,0).

See Also

IDirectSound3DListener::SetVelocity

IDirectSound3DListener::SetAllParameters

The **IDirectSound3DListener::SetAllParameters** method sets all 3-D listener parameters from a given **DS3DLISTENER** structure that describes all aspects of the 3-D listener at a moment in time.

```
HRESULT SetAllParameters(
  LPCDS3DLISTENER lpcListener,
  DWORD dwApply
);
```

Parameters

lpcListener

Address of a **DS3DLISTENER** structure that contains information describing all current 3-D listener parameters.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

See Also

IDirectSound3DListener::GetAllParameters

IDirectSound3DListener::SetDistanceFactor

The **IDirectSound3DListener::SetDistanceFactor** method sets the current distance factor.

```
HRESULT SetDistanceFactor(
  D3DVALUE flDistanceFactor,
  DWORD dwApply
);
```

Parameters

flDistanceFactor
New distance factor.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

For additional information about distance factors, see Distance Factor.

See Also

IDirectSound3DListener::GetDistanceFactor

IDirectSound3DListener::SetDopplerFactor

The **IDirectSound3DListener::SetDopplerFactor** method sets the current Doppler effect factor.

```
HRESULT SetDopplerFactor(
  D3DVALUE flDopplerFactor,
  DWORD dwApply
);
```

Parameters

flDopplerFactor
New Doppler factor value.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The Doppler factor has a range of DS3D_MINDOPPLERFACTOR (no Doppler effects) to DS3D_MAXDOPPLERFACTOR (as currently defined, 10 times the Doppler effects found in the real world). The default value is DS3D_DEFAULTDOPPLERFACTOR (1.0). For additional information, see Doppler Factor.

See Also

IDirectSound3DListener::GetDopplerFactor

IDirectSound3DListener::SetOrientation

The **IDirectSound3DListener::SetOrientation** method sets the listener's current orientation in terms of two vectors: a front vector and a top vector.

```
HRESULT SetOrientation(
  D3DVALUE xFront,
  D3DVALUE yFront,
  D3DVALUE zFront,
  D3DVALUE xTop,
  D3DVALUE yTop,
  D3DVALUE zTop,
  DWORD dwApply
);
```

Parameters

xFront, yFront, and *zFront*
Values whose types are **D3DVALUE** and that represent the coordinates of the front orientation vector.

xTop, yTop, and *zTop*
Values whose types are **D3DVALUE** and that represent the coordinates of the top orientation vector.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The front vector points in the direction of the listener's nose, and the top vector points out the top of the listener's head. By default, the front vector is (0,0,1.0) and the top vector is (0,1.0,0).

See Also

IDirectSound3DListener::GetOrientation

IDirectSound3DListener::SetPosition

The **IDirectSound3DListener::SetPosition** method sets the listener's current position, in distance units. By default, these units are meters, but this can be changed by calling the **IDirectSound3DListener::SetDistanceFactor** method.

```
HRESULT SetPosition(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*

Values whose types are **D3DVALUE** and that represent the coordinates of the listener's new position vector.

dwApply

Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

See Also

IDirectSound3DListener::GetPosition

IDirectSound3DListener::SetRolloffFactor

The **IDirectSound3DListener::SetRolloffFactor** method sets the rolloff factor.

```
HRESULT SetRolloffFactor(
  D3DVALUE flRolloffFactor,
  DWORD dwApply
);
```

Parameters

flRolloffFactor
New rolloff factor.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

The rolloff factor has a range of DS3D_MINROLLOFFFACTOR (no rolloff) to DS3D_MAXROLLOFFFACTOR (as currently defined, 10 times the rolloff found in the real world). The default value is DS3D_DEFAULTROLLOFFFACTOR (1.0). For additional information, see Rolloff Factor.

See Also

IDirectSound3DListener::GetRolloffFactor

IDirectSound3DListener::SetVelocity

The **IDirectSound3DListener::SetVelocity** method sets the listener's velocity.

```
HRESULT SetVelocity(
  D3DVALUE x,
  D3DVALUE y,
  D3DVALUE z,
  DWORD dwApply
);
```

Parameters

x, y, and *z*
Values whose types are **D3DVALUE** and that represent the coordinates of the listener's new velocity vector.

dwApply
Value indicating when the setting should be applied. This value must be one of the following:

DS3D_DEFERRED	Settings are not applied until the application calls the **IDirectSound3DListener::CommitDeferredSettings** method. This allows the application to change several settings and generate a single recalculation.
DS3D_IMMEDIATE	Settings are applied immediately, causing the system to recalculate the 3-D coordinates for all 3-D sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

Velocity is used only for Doppler effects. It does not actually move the listener. To change the listener's position, use the **IDirectSound3DListener::SetPosition** method. The default velocity is (0,0,0).

See Also

IDirectSound3DListener::GetVelocity

IDirectSoundBuffer

Applications use the methods of the **IDirectSoundBuffer** interface to create DirectSoundBuffer objects and set up the environment.

The interface is obtained by using the **IDirectSound::CreateSoundBuffer** method.

The **IDirectSoundBuffer** methods can be organized into the following groups:

Information	**GetCaps**
	GetFormat
	GetStatus
	SetFormat
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. This interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUNDBUFFER** type is defined as a pointer to the **IDirectSoundBuffer** interface:

```
typedef struct IDirectSoundBuffer    *LPDIRECTSOUNDBUFFER;
```

IDirectSoundBuffer::GetCaps

The **IDirectSoundBuffer::GetCaps** method retrieves the capabilities of the DirectSoundBuffer object.

```
HRESULT GetCaps(
  LPDSBCAPS lpDSBufferCaps
);
```

Parameters

lpDSBufferCaps
Address of a **DSBCAPS** structure to contain the capabilities of this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

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 buffer's location, 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.

See Also

DSBCAPS, **DSBUFFERDESC**, **IDirectSoundBuffer**, **IDirectSound::CreateSoundBuffer**

IDirectSoundBuffer::GetCurrentPosition

The **IDirectSoundBuffer::GetCurrentPosition** method retrieves the current position of the play and write cursors in the sound buffer.

```
HRESULT GetCurrentPosition(
  LPDWORD lpdwCurrentPlayCursor,
  LPDWORD lpdwCurrentWriteCursor
);
```

Parameters

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. This parameter can be NULL if the current play position is not wanted.

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. This parameter can be NULL if the current write position is not wanted.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The write cursor indicates the position at which it is safe to write new data to the buffer. The write cursor always leads the play cursor, typically by about 15 milliseconds' worth of audio data. For more information, see Current Play and Write Positions.

It is always safe to change data that is behind the position indicated by the *lpdwCurrentPlayCursor* parameter.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetCurrentPosition**

IDirectSoundBuffer::GetFormat

The **IDirectSoundBuffer::GetFormat** method retrieves a description of the format of the sound data in the buffer, or the buffer size needed to retrieve the format description.

```
HRESULT GetFormat(
  LPWAVEFORMATEX lpwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD lpdwSizeWritten
);
```

Parameters

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. In this case the **DWORD** pointed to by the *lpdwSizeWritten* parameter will receive the size of the structure needed to receive complete format information.

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.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

Remarks

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::GetFrequency

The **IDirectSoundBuffer::GetFrequency** method retrieves the frequency, in samples per second, at which the buffer is playing.

```
HRESULT GetFrequency(
  LPDWORD lpdwFrequency
);
```

Parameters

lpdwFrequency
Address of the variable that represents the frequency at which the audio buffer is being played.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The frequency value will be in the range of DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are currently defined in Dsound.h as 100 and 100,000 respectively.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetFrequency**

IDirectSoundBuffer::GetPan

The **IDirectSoundBuffer::GetPan** method retrieves a variable that represents the relative volume between the left and right audio channels.

```
HRESULT GetPan(
  LPLONG lplPan
);
```

Parameters

lplPan
Address of a variable to contain the relative mix between the left and right speakers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The returned value is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as -10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the *lplPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, 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 DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT 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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::GetStatus

The **IDirectSoundBuffer::GetStatus** method retrieves the current status of the sound buffer.

```
HRESULT GetStatus(
  LPDWORD lpdwStatus
);
```

Parameters

lpdwStatus

Address of a variable to contain the status of the sound buffer. The status can be a combination of the following flags:

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 playing. If this value is not set, the buffer is stopped.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

See Also

IDirectSoundBuffer

IDirectSoundBuffer::GetVolume

The **IDirectSoundBuffer::GetVolume** method retrieves the current volume for this sound buffer.

```
HRESULT GetVolume(
  LPLONG lplVolume
);
```

Parameters

lplVolume
Address of the variable to contain the volume associated with the specified DirectSound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::Initialize

The **IDirectSoundBuffer::Initialize** method initializes a DirectSoundBuffer object if it has not yet been initialized.

```
HRESULT Initialize(
  LPDIRECTSOUND lpDirectSound,
  LPCDSBUFFERDESC lpcDSBufferDesc
);
```

Parameters

lpDirectSound
Address of the DirectSound object associated with this DirectSoundBuffer object.

lpcDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains the values used to initialize this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values.

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

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.

See Also

DSBUFFERDESC, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer**

IDirectSoundBuffer::Lock

The **IDirectSoundBuffer::Lock** method obtains a valid write pointer to the sound buffer's audio data.

```
HRESULT Lock(
  DWORD dwWriteCursor,
  DWORD dwWriteBytes,
  LPVOID lplpvAudioPtr1,
  LPDWORD lpdwAudioBytes1,
  LPVOID lplpvAudioPtr2,
  LPDWORD lpdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

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 flags are defined:

DSBLOCK_FROMWRITECURSOR
Locks from the current write position, making a call to **IDirectSoundBuffer::GetCurrentPosition** unnecessary. If this flag is specified, the *dwWriteCursor* parameter is ignored.

DSBLOCK_ENTIREBUFFER
Locks the entire buffer. The *dwWriteBytes* parameter is ignored.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM

DSERR_PRIOLEVELNEEDED

Remarks

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 to 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 might not be valid even though the DirectSoundBuffer object contains valid sound data. For example, if the buffer is located in on-board memory, the pointer might be an address to a temporary buffer in main system memory. When **IDirectSoundBuffer::Unlock** is called, this temporary buffer will be transferred to the on-board memory.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Unlock**

IDirectSoundBuffer::Play

The **IDirectSoundBuffer::Play** method causes the sound buffer to play from the current position.

```
HRESULT Play(
  DWORD dwReserved1,
  DWORD dwReserved2,
  DWORD dwFlags
);
```

Parameters

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 sound buffers.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

This method will cause a secondary sound buffer to be mixed into the primary buffer and sent 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 need not 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 defined in the most recent call supersede flags defined in previous calls.

Primary buffers must be played with the DSBPLAY_LOOPING flag set.

This method will cause primary sound buffers 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 sent 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 can reduce processing overhead when sounds are started and stopped in sequence, because 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 should 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 with DS_OK, but no sound will be produced until **IDirectSound::SetCooperativeLevel** is called.

See Also

IDirectSoundBuffer, **IDirectSound::SetCooperativeLevel**

IDirectSoundBuffer::Restore

The **IDirectSoundBuffer::Restore** method restores the memory allocation for a lost sound buffer for the specified DirectSoundBuffer object.

```
HRESULT Restore();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BUFFERLOST
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

If the application does not have the input focus, **IDirectSoundBuffer::Restore** might 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 sound 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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Lock**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::GetStatus**

IDirectSoundBuffer::SetCurrentPosition

The **IDirectSoundBuffer::SetCurrentPosition** method moves the current play position for secondary sound buffers.

```
HRESULT SetCurrentPosition(
  DWORD dwNewPosition
);
```

Parameters

dwNewPosition

New position, in bytes, from the beginning of the buffer that will be used when the sound buffer is played.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::SetFormat

The **IDirectSoundBuffer::SetFormat** method 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.

```
HRESULT SetFormat(
  LPCWAVEFORMATEX lpcfxFormat
);
```

Parameters

lpcfxFormat
Address of a **WAVEFORMATEX** structure that describes the new format for the primary sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_BADFORMAT
DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY
DSERR_PRIOLEVELNEEDED
DSERR_UNSUPPORTED

Remarks

If this method is called on a primary buffer that is being accessed in write-primary cooperative level, the buffer must be stopped before **IDirectSoundBuffer::SetFormat** is called. If this method is being called on a primary buffer for a non-write-primary level, DirectSound will implicitly stop the primary buffer, change the format, and restart the primary; the application need not do this explicitly.

If the hardware does not support the requested format, DirectSound may be able to mix the sound in the requested format and then convert it before sending it to the primary buffer. To determine whether this is happening, an application can call the **IDirectSoundBuffer::GetFormat** method for the primary buffer and compare the result with the format that was requested with the **SetFormat** method.

If the hardware does not support the requested format and DirectSound is unable to emulate it, the call to **SetFormat** fails.

A call to this method also fails if the calling application has the DSSCL_NORMAL cooperative level.

This method is not valid for secondary sound buffers. If a secondary sound 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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetFormat**

IDirectSoundBuffer::SetFrequency

The **IDirectSoundBuffer::SetFrequency** method sets the frequency at which the audio samples are played.

```
HRESULT SetFrequency(
  DWORD dwFrequency
);
```

Parameters

dwFrequency

New frequency, in hertz (Hz), at which to play the audio samples. The value must be in the range DSBFREQUENCY_MIN to DSBFREQUENCY_MAX. These values are currently defined in Dsound.h as 100 and 100,000 respectively.

If the value is DSBFREQUENCY_ORIGINAL, the frequency is reset to the default value in the current buffer format. This format is specified in the **IDirectSound::CreateSoundBuffer** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

Increasing or decreasing the frequency changes the perceived pitch of the audio data. This method does not affect the format of the buffer.

This method is not valid for primary sound buffers.

See Also

IDirectSoundBuffer, **IDirectSound::CreateSoundBuffer**, **IDirectSoundBuffer::GetFrequency**, **IDirectSoundBuffer::Play**, **IDirectSoundBuffer::SetFormat**

IDirectSoundBuffer::SetPan

The **IDirectSoundBuffer::SetPan** method specifies the relative volume between the left and right channels.

```
HRESULT SetPan(
  LONG lPan
);
```

Parameters

lPan
Relative volume between the left and right channels.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The value in *lPan* is measured in hundredths of a decibel (dB), in the range of DSBPAN_LEFT to DSBPAN_RIGHT. These values are currently defined in Dsound.h as -10,000 and 10,000 respectively. The value DSBPAN_LEFT means the right channel is attenuated by 100 dB. The value DSBPAN_RIGHT means the left channel is attenuated by 100 dB. The neutral value is DSBPAN_CENTER, defined as zero. This value of 0 in the *lplPan* parameter means that both channels are at full volume (they are attenuated by 0 decibels). At any setting other than DSBPAN_CENTER, 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 DSBPAN_LEFT means that the right channel is silent and the sound is all the way to the left, while a pan of DSBPAN_RIGHT 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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetVolume**

IDirectSoundBuffer::SetVolume

The **IDirectSoundBuffer::SetVolume** method changes the volume of a sound buffer.

```
HRESULT SetVolume(
  LONG lVolume
);
```

Parameters

lVolume
New volume requested for this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_CONTROLUNAVAIL
DSERR_GENERIC
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

The volume is specified in hundredths of decibels (dB). Allowable values are between DSBVOLUME_MAX (no attenuation) and DSBVOLUME_MIN (silence). These values are currently defined in Dsound.h as 0 and -10,000 respectively. The value DSBVOLUME_MAX represents the original, unadjusted volume of the stream. The value DSBVOLUME_MIN indicates an audio volume attenuated by 100 dB, which, for all practical purposes, is silence. Currently DirectSound does not support amplification.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetPan**, **IDirectSoundBuffer::GetVolume**, **IDirectSoundBuffer::SetPan**

IDirectSoundBuffer::Stop

The **IDirectSoundBuffer::Stop** method causes the sound buffer to stop playing.

```
HRESULT Stop();
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

For secondary sound 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 sound 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 play only 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 decibels.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::Play**

IDirectSoundBuffer::Unlock

The **IDirectSoundBuffer::Unlock** method releases a locked sound buffer.

```
HRESULT Unlock(
  LPVOID lpvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID lpvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

lpvAudioPtr1
Address of the value retrieved in the *lplpvAudioPtr1* parameter of the **IDirectSoundBuffer::Lock** method.

dwAudioBytes1
Number of bytes actually written to 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 to the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundBuffer::Lock** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDCALL
DSERR_INVALIDPARAM
DSERR_PRIOLEVELNEEDED

Remarks

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.

See Also

IDirectSoundBuffer, **IDirectSoundBuffer::GetCurrentPosition**, **IDirectSoundBuffer::Lock**

IDirectSoundCapture

The methods of the **IDirectSoundCapture** interface are used to create sound capture buffers.

The interface is obtained by using the **DirectSoundCaptureCreate** function.

This reference section gives information on the following methods of the **IDirectSoundCapture** interface:

CreateCaptureBuffer

GetCaps

Initialize

Like all COM interfaces, the **IDirectSoundCapture** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUNDCAPTURE** type is defined as a pointer to the **IDirectSoundCapture** interface:

```
typedef struct IDirectSoundCapture    *LPDIRECTSOUNDCAPTURE;
```

IDirectSoundCapture::CreateCaptureBuffer

The **IDirectSoundCapture::CreateCaptureBuffer** method creates a capture buffer.

Unlike DirectSound, which can mix several sounds into one sound for output, DirectSoundCapture cannot do the exact opposite and extract various sounds from one input sound. For the first version, DirectSoundCapture allows only one capture buffer to exist at any given time per capture device.

```
HRESULT IDirectSoundCapture::CreateCaptureBuffer(
  LPDSCBUFFERDESC lpDSCBufferDesc,
  LPLPDIRECTSOUNDCAPTUREBUFFER lplpDirectSoundCaptureBuffer,
  LPUNKNOWN pUnkOuter
);
```

Parameters

lpDSCBufferDesc
Pointer to a **DSCBUFFERDESC** structure containing values for the capture buffer being created.

lplpDirectSoundCaptureBuffer
Address of the **IDirectSoundCaptureBuffer** interface pointer if successful.

punkOuter
Controlling **IUnknown** of the aggregate. Its value must be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_BADFORMAT
DSERR_GENERIC
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

IDirectSoundCapture::GetCaps

The **IDirectSoundCapture::GetCaps** method obtains the capabilities of the capture device.

```
HRESULT IDirectSoundCapture::GetCaps(
  LPDSCCAPS lpDSCCaps
);
```

Parameters

lpDSCCaps

Pointer to a **DSCCAPS** structure to be filled by the capture device. When the method is called, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_UNINITIALIZED

IDirectSoundCapture::Initialize

When **CoCreateInstance** is used to create a DirectSoundCapture object, the object must be initialized with the **IDirectSoundCapture::Initialize** method. Calling this method is not required when the **DirectSoundCaptureCreate** function is used to create the object.

```
HRESULT Initialize(
  LPGUID lpGuid
);
```

Parameters

lpGuid

Address of the GUID specifying the sound driver for the DirectSoundCapture object to bind to. Use NULL to select the primary sound driver.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY
DSERR_ALREADYINITIALIZED

IDirectSoundCaptureBuffer

The methods of the **IDirectSoundCaptureBuffer** interface are used to manipulate sound capture buffers.

The interface is obtained by calling the **IDirectSoundCapture::CreateCaptureBuffer** method.

The methods of the **IDirectSoundCaptureBuffer** interface may be grouped as follows:

Information	**GetCaps**
	GetCurrentPosition
	GetFormat
	GetStatus
Capture management	**Lock**
	Start
	Stop
	Unlock

Like all COM interfaces, the **IDirectSoundCaptureBuffer** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUNDCAPTUREBUFFER** type is defined as a pointer to the **IDirectSoundCaptureBuffer** interface:

```
typedef struct IDirectSoundCaptureBuffer  *LPDIRECTSOUNDCAPTUREBUFFER;
```

IDirectSoundCaptureBuffer::GetCaps

The **IDirectSoundCaptureBuffer::GetCaps** method returns the capabilities of the DirectSound Capture Buffer.

```
HRESULT GetCaps(
  LPDSCBCAPS lpDSCBCaps
);
```

Parameters

lpDSCBCaps

Pointer to a **DSCBCAPS** structure to be filled by the capture buffer. On input, the **dwSize** member must specify the size of the structure in bytes.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_UNSUPPORTED
DSERR_OUTOFMEMORY

IDirectSoundCaptureBuffer::GetCurrentPosition

The **IDirectSoundCaptureBuffer::GetCurrentPosition** method gets the current capture and reads position in the buffer.

The capture position is ahead of the read position. These positions are not always identical due to possible buffering of captured data either on the physical device or in the host. The data after the read position up to and including the capture position is not necessarily valid data.

```
HRESULT GetCurrentPosition(
  LPDWORD lpdwCapturePosition,
  LPDWORD lpdwReadPosition
);
```

Parameters

lpdwCapturePosition

Address of a variable to receive the current capture position in the DirectSoundCaptureBuffer object. This position is an offset within the capture buffer and is specified in bytes. The value can be NULL if the caller is not interested in this position information.

lpdwReadPosition

Address of a variable to receive the current position in the DirectSoundCaptureBuffer object at which it is safe to read data. This position is an offset within the capture buffer and is specified in bytes. The value can be NULL if the caller is not interested in this position information.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

IDirectSoundCaptureBuffer::GetFormat

The **IDirectSoundCaptureBuffer::GetFormat** method retrieves the current format of the sound capture buffer.

```
HRESULT IDirectSoundCaptureBufferGetFormat(
  LPWAVEFORMATEX lpwfxFormat,
  DWORD dwSizeAllocated,
  LPDWORD lpdwSizeWritten
);
```

Parameters

lpwfxFormat

Address of the **WAVEFORMATEX** variable to contain a description of the sound data in the capture buffer. To retrieve the buffer size needed to contain the format description, specify NULL. In this case the **DWORD** pointed to by the *lpdwSizeWritten* parameter will receive the size of the structure needed to receive complete format information.

dwSizeAllocated

Size, in bytes, of the **WAVEFORMATEX** structure. DirectSoundCapture writes, at most, **dwSizeAllocated** bytes to the structure; if the structure requires more memory, it is truncated.

lpdwSizeWritten

Address of a variable to receive the number of bytes written to the **WAVEFORMATEX** structure. This parameter can be NULL.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSoundCaptureBuffer::GetStatus

The **IDirectSoundCaptureBuffer::GetStatus** method retrieves the current status of the sound capture buffer.

```
HRESULT IDirectSoundCaptureBuffer::GetStatus(
  DWORD *lpdwStatus
);
```

Parameters

lpdwStatus

Address of a variable to contain the status of the sound capture buffer. The status can be set to one or more of the following:

DSCBSTATUS_CAPTURING
DSCBSTATUS_LOOPING

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be DSERR_INVALIDPARAM.

IDirectSoundCaptureBuffer::Initialize

The **IDirectSoundCaptureBuffer::Initialize** method initializes a DirectSoundCaptureBuffer object if it has not yet been initialized.

```
HRESULT Initialize(
  LPDIRECTSOUNDCAPTURE lpDirectSoundCapture,
  LPCDSBUFFERDESC lpcDSBufferDesc
);
```

Parameters

lpDirectSoundCapture
Address of the DirectSoundCapture object associated with this DirectSoundCaptureBuffer object.

lpcDSBufferDesc
Address of a **DSBUFFERDESC** structure that contains the values used to initialize this sound buffer.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_ALREADYINITIALIZED

Remarks

Because the **IDirectSoundCapture::CreateCaptureBuffer** method calls the **IDirectSoundCaptureBuffer::Initialize** method internally, it is not needed for the current release of DirectSound. This method is provided for future extensibility.

See Also

DSBUFFERDESC, **IDirectSoundCapture::CreateCaptureBuffer**

IDirectSoundCaptureBuffer::Lock

The **IDirectSoundCaptureBuffer::Lock** method locks the buffer. Locking the buffer returns pointers into the buffer, allowing the application to read or write audio data into that memory.

This method accepts an offset and a byte count, and returns two read pointers and their associated sizes. Two pointers are required because sound capture 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, DirectSoundCapture will not lock the wraparound portion of the buffer.

The application should read data from the pointers returned by the **IDirectSoundCaptureBuffer::Lock** method and then call the **IDirectSoundCaptureBuffer::Unlock** method to release the buffer back to DirectSoundCapture. The sound buffer should not be locked for long periods of time; if it is, the capture cursor will reach the locked bytes and configuration-dependent audio problems may result.

```
HRESULT IDirectSoundCaptureBuffer::Lock(
  DWORD dwReadCursor,
  DWORD dwReadBytes,
  LPVOID *lplpvAudioPtr1,
  LPDWORD lpdwAudioBytes1,
  LPVOID *lplpvAudioPtr2,
  LPDWORD lpdwAudioBytes2,
  DWORD dwFlags
);
```

Parameters

dwReadCursor
Offset, in bytes, from the start of the buffer to where the lock begins.

dwReadBytes
Size, in bytes, of the portion of the buffer to lock. Note that the sound capture buffer is conceptually circular.

lplpvAudioPtr1
Address of a pointer to contain the first block of the sound capture 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 *dwReadBytes* parameter, *lplpvAudioPtr2* will point to a second block of data.

lplpvAudioPtr2
Address of a pointer to contain the second block of the sound capture buffer to be locked. If the value of this parameter is NULL, the *lplpvAudioPtr1* parameter points to the entire locked portion of the sound capture 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. This value can be NULL or the following flag:

DSCBLOCK_ENTIREBUFFER
The *dwReadBytes* parameter is to be ignored and the entire

capture buffer is to be locked.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following values:

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

IDirectSoundCaptureBuffer::Start

The **IDirectSoundCaptureBuffer::Start** method puts the capture buffer into the capture state and begins capturing data into the buffer. If the capture buffer is already in the capture state then the method has no effect.

```
HRESULT IDirectSoundCaptureBuffer::Start(
  DWORD dwFlags
);
```

Parameters

dwFlags

Flags that specify the behavior for the capture buffer when capturing sound data. Possible values for **dwFlags** can be one of the following:

DSCBSTART_LOOPING

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_NODRIVER
DSERR_OUTOFMEMORY

IDirectSoundCaptureBuffer::Stop

The **IDirectSoundCaptureBuffer::Stop** method puts the capture buffer into the “stop” state and stops capturing data. If the capture buffer is already in the stop state then the method has no effect.

```
HRESULT IDirectSoundCaptureBuffer::Stop(
  void
);
```

Parameters

None.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_NODRIVER
DSERR_OUTOFMEMORY

IDirectSoundCaptureBuffer::Unlock

The **IDirectSoundCaptureBuffer::Unlock** method unlocks the buffer.

An application must pass both pointers, *lpvAudioPtr1* and *lpvAudioPtr2*, returned by the **IDirectSoundCaptureBuffer::Lock** method to ensure the correct pairing of **IDirectSoundCaptureBuffer::Lock** and **IDirectSoundCaptureBuffer::Unlock**. The second pointer is needed even if zero 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 that the sound capture buffer does not remain locked for long periods of time.

```
HRESULT Unlock(
  LPVOID lpvAudioPtr1,
  DWORD dwAudioBytes1,
  LPVOID lpvAudioPtr2,
  DWORD dwAudioBytes2
);
```

Parameters

lpvAudioPtr1
Address of the value retrieved in the *lplpvAudioPtr1* parameter of the IDirectSoundCaptureBuffer::Lock method.

dwAudioBytes1
Number of bytes actually read from the *lpvAudioPtr1* parameter. It should not exceed the number of bytes returned by the **IDirectSoundCaptureBuffer::Lock** method.

lpvAudioPtr2
Address of the value retrieved in the *lplpvAudioPtr2* parameter of the **IDirectSoundCaptureBuffer::Lock** method.

dwAudioBytes2
Number of bytes actually read from the *lpvAudioPtr2* parameter. It should not exceed the number of bytes returned by the **IDirectSoundCaptureBuffer::Lock** method.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following values:

DSERR_INVALIDPARAM
DSERR_INVALIDCALL

IDirectSoundNotify

The **IDirectSoundNotify** interface provides a mechanism for setting up notification events for a playback or capture buffer.

The interface is obtained by calling the **QueryInterface** method of an existing interface on a DirectSoundBuffer object. For an example, see Play Buffer Notification.

The interface has the following method:

SetNotificationPositions

Like all COM interfaces, the **IDirectSoundNotify** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECTSOUNDNOTIFY** type is defined as a pointer to the **IDirectSoundNotify** interface:

```
typedef struct IDirectSoundNotify    *LPDIRECTSOUNDNOTIFY;
```

IDirectSoundNotify::SetNotificationPositions

The **IDirectSoundCaptureBuffer::SetNotificationPositions** method sets the notification positions. During capture or playback, whenever the position reaches an offset specified in one of the **DSBPOSITIONNOTIFY** structures in the caller-supplied array, the associated event will be signaled. The position tracked in playback is the current play position; in capture it is the current read position.

```
HRESULT IDirectSoundNotify::SetNotificationPositions(
  DWORD cPositionNotifies,
  LPCDSBPOSITIONNOTIFY lpcPositionNotifies
);
```

Parameters

cPositionNotifies
Number of **DSBPOSITIONNOTIFY** structures.

lpcPositionNotifies
Pointer to an array of **DSBPOSITIONNOTIFY** structures.

Return Values

If the method succeeds, the return value is DS_OK.

If the method fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_OUTOFMEMORY

Remarks

The value DSBPN_OFFSETSTOP can be specified in the **dwOffset** member to tell DirectSound to signal the associated event when the **IDirectSoundBuffer::Stop** or **IDirectSoundCaptureBuffer::Stop** method is called or when the end of the buffer has been reached and the playback is not looping. If it is used, this should be the last item in the position-notify array.

If a position-notify array has already been set, calling this function again will replace the previous position-notify array.

The buffer must be stopped when this method is called.

IKsPropertySet

The **IKsPropertySet** interface is used to get information about extended properties of sound devices, and to manipulate those properties.

The interface is obtained by calling the **QueryInterface** method of an existing interface on a DirectSoundBuffer object. For more information, see DirectSound Property Sets.

The **IKsPropertyS**et method has the following methods:

Get
QuerySupport
Set

Like all COM interfaces, the **IKsPropertySet** interface also inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPKSPROPERTYSET** type is defined as a pointer to the **IKsPropertySet** interface:

```
typedef struct IKsPropertySet    *LPKSPROPERTYSET;
```

IKsPropertySet::Get

The **IKsPropertySet::Get** method retrieves the current value of a property. This method copies the value of the property to the buffer pointed to by *pPropertyValue*.

```
HRESULT Get(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PVOID pPropertyParams,
  ULONG cbPropertyParams,
  PVOID pPropertyData,
  ULONG cbPropertyData,
  PULONG pcbReturnedData
);
```

Parameters

Proper tySetId
PropertyId
The property within the property set.

pPropertyParams
Pointer to instance parameters for the property.

cbPropertyParams
The number of bytes in the structure pointed to by *pPropertyParams*.

pPropertyData
Pointer to buffer in which to store the value of the property.

cbPropertyData
The number of bytes in the structure pointed to by *pPropertyData*.

pcbReturnedData
The number of bytes returned in the structure pointed to by *pPropertyData*.

Return Values

Return values are determined by the designer of the property set.

IKsPropertySet::QuerySupport

The **IKsPropertySet::QuerySupport** method determines whether a property set is supported by the object.

```
HRESULT QuerySetSupport(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PULONG pSupport
);
```

Parameters

PropertySetId
Identifier of the property set.

PropertyId
Identifier of the property within the property set.

pSupport
Pointer to a **ULONG** in which to store flags indicating the support provided by the driver. The following values may be set:

KSPROPERTY_SUPPORT_GET
The driver sets this flag to indicate that the property can be read by using the **IKsPropertySet::Get** method.

KSPROPERTY_SUPPORT_SET
The driver sets this flag to indicate that the property can be changed by using the **IKsPropertySet::Set** method.

Return Values

Returns E_NOTIMPL for property sets that are not supported. Other return values are determined by the designer of the property set.

Remarks

Whether it is valid to support some properties within the set but not others depends on the definition of the property set. Consult the hardware manufacturer's specification for the property set of interest.

IKsPropertySet::Set

The **IKsPropertySet::SetProperty** method sets the current value of a property to the value stored in the buffer pointed to by *pPropertyValue*.

```
HRESULT SetProperty(
  REFGUID PropertySetId,
  ULONG PropertyId,
  PVOID pPropertyParams,
  ULONG cbPropertyParams,
  PVOID pPropertyData,
  ULONG cbPropertyData
);
```

Parameters

PropertySetId
The property set.

PropertyId
The property within the property set.

pPropertyParams
Pointer to instance parameters for the property.

cbPropertyParams
Size of the structure pointed to by *pPropertyParams*.

pPropertyData
Address of a buffer containing the value to which to set the property.

cbPropertyData
Size of the structure pointed to by *pPropertyData*.

Return Values

Return values are determined by the designer of the property set.

Functions

This section contains reference information for the following DirectSound and DirectSoundCapture global functions:

- **DirectSoundCreate**
- **DirectSoundEnumerate**
- **DirectSoundCaptureCreate**
- **DirectSoundCaptureEnumerate**

DirectSoundCreate

The **DirectSoundCreate** function creates and initializes an **IDirectSound** interface.

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

Parameters

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 NULL for the default device.

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.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be one of the following error values:

DSERR_ALLOCATED
DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_NODRIVER
DSERR_OUTOFMEMORY

Remarks

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

See Also

IDirectSound::GetCaps, **IDirectSound::SetCooperativeLevel**

DirectSoundEnumerate

The **DirectSoundEnumerate** function enumerates the DirectSound drivers installed in the system.

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

Parameters

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.

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be DSERR_INVALIDPARAM.

See Also

DSEnumCallback

DirectSoundCaptureCreate

The **DirectSoundCaptureCreate** function creates and initializes an object that supports the **IDirectSoundCapture** interface.

```
HRESULT DirectSoundCaptureCreate(
  LPGUID lpGUID,
  LPDIRECTSOUNDCAPTURE *lplpDSC,
  LPUNKNOWN pUnkOuter
);
```

Parameters

lpGUID
Address of the GUID that identifies the sound capture device. The value of this parameter must be one of the GUIDs returned by **DirectSoundCaptureEnumerate**, or NULL for the default device.

lplpDSC
Address of a pointer to a DirectSoundCapture object created in response to this function.

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

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be one of the following error values:

DSERR_INVALIDPARAM
DSERR_NOAGGREGATION
DSERR_OUTOFMEMORY

DirectSoundCaptureEnumerate

The **DirectSoundCaptureEnumerate** function enumerates the DirectSoundCapture objects installed in the system.

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

Parameters

lpDSEnumCallback

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

lpContext

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

Return Values

If the function succeeds, the return value is DS_OK.

If the function fails, the return value may be DSERR_INVALIDPARAM.

Callback Function

This section contains reference information for the following DirectSound callback function.

- **DSEnumCallback**

DSEnumCallback

DSEnumCallback is an 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** or **DirectSoundCaptureEnumerate** function.

```
BOOL DSEnumCallback(
  LPGUID lpGuid,
  LPCSTR lpstrDescription,
  LPCSTR lpstrModule,
  LPVOID lpContext
);
```

Parameters

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.

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

lpcstrModule
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.

Return Values

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

Remarks

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

See Also

DirectSoundEnumerate

Structures

This section contains reference information for the following structures used with DirectSound:

- **DS3DBUFFER**
- **DS3DLISTENER**
- **DSBCAPS**
- **DSBPOSITIONNOTIFY**
- **DSBUFFERDESC**
- **DSCAPS**
- **DSCBCAPS**
- **DSCBUFFERDESC**
- **DSCCAPS**

DS3DBUFFER

The **DS3DBUFFER** structure contains all information necessary to uniquely describe the location, orientation, and motion of a 3-D sound buffer. This structure is used with the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods.

```
typedef struct {
    DWORD     dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    DWORD     dwInsideConeAngle;
    DWORD     dwOutsideConeAngle;
    D3DVECTOR  vConeOrientation;
    LONG      lConeOutsideVolume;
    D3DVALUE     flMinDistance;
    D3DVALUE     flMaxDistance;
    DWORD     dwMode;
} DS3DBUFFER, *LPDS3DBUFFER;

typedef const DS3DBUFFER *LPCDS3DBUFFER;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

vPosition
A **D3DVECTOR** structurethat describes the current position of the 3-D sound buffer.

vVelocity
A **D3DVECTOR** structure that describes the current velocity of the 3-D sound buffer.

dwInsideConeAngle
The angle of the inside sound projection cone.

dwOutsideConeAngle
The angle of the outside sound projection cone.

vConeOrientation
A **D3DVECTOR** structure that describes the current orientation of this 3-D buffer's sound projection cone.

lConeOutsideVolume
The cone outside volume.

flMinDistance
The minimum distance.

flMaxDistance
The maximum distance.

dwMode
The 3-D sound processing mode to be set.

DS3DMODE_DISABLE
3-D sound processing is disabled. The sound will appear to originate from the center of the listener's head.

DS3DMODE_HEADRELATIVE
Sound parameters (position, velocity, and orientation) are

relative to the listener's parameters. In this mode, the absolute parameters of the sound are updated automatically as the listener's parameters change, so that the relative parameters remain constant.

DS3DMODE_NORMAL

Normal processing. This is the default mode.

DS3DLISTENER

The **DS3DLISTENER** structure contains all information necessary to uniquely describe the 3-D world parameters and position of the listener. This structure is used with the **IDirectSound3DListener::GetAllParameters** and **IDirectSound3DListener::SetAllParameters** methods.

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    D3DVECTOR  vOrientFront;
    D3DVECTOR  vOrientTop;
    D3DVALUE   flDistanceFactor;
    D3DVALUE   flRolloffFactor;
    D3DVALUE   flDopplerFactor;
} DS3DLISTENER, *LPDS3DLISTENER;

typedef const DS3DLISTENER *LPCDS3DLISTENER;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

vPosition, **vVelocity**, **vOrientFront**, and **vOrientTop**
D3DVECTOR structures that describe the listener's position, velocity, front orientation, and top orientation, respectively.

flDistanceFactor, **flRolloffFactor**, and **flDopplerFactor**
The current distance, rolloff, and Doppler factors, respectively.

DSBCAPS

The **DSBCAPS** structure specifies the capabilities of a DirectSound buffer object, for use by the **IDirectSoundBuffer::GetCaps** method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwBufferBytes;
    DWORD dwUnlockTransferRate;
    DWORD dwPlayCpuOverhead;
} DSBCAPS, *LPDSBCAPS;

typedef const DSBCAPS *LPCDSBCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Flags that specify buffer-object capabilities.

DSBCAPS_CTRL3D

The buffer is either a primary buffer or a secondary buffer that uses 3-D control. To create a primary buffer, the **dwFlags** member of the **DSBUFFERDESC** structure should include the **DSBCAPS_PRIMARYBUFFER** flag.

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_GETCURRENTPOSITION2

Indicates that **IDirectSoundBuffer::GetCurrentPosition** should use the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was significantly ahead of the actual playing sound on emulated sound cards; it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play position. If this flag is not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards; if a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS

The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one exception is if you switch

focus to a DirectSound application that uses the DSSCL_EXCLUSIVE or DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

DSBCAPS_LOCHARDWARE

The buffer is in hardware memory and uses hardware mixing.

DSBCAPS_LOCSOFTWARE

The buffer is in software memory and uses software mixing.

DSBCAPS_MUTE3DATMAXDISTANCE

The sound is reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER

Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary sound 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 are muted, but the sticky focus buffers are still audible. This is useful for nongame applications, such as movie playback (DirectShow™), when the user wants to hear the soundtrack while typing in Microsoft Word or Microsoft Excel, for example. However, if the user switches to another DirectSound application, all sound buffers, both normal and sticky focus, in the previous application are 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 because no processing is required. For hardware buffers, the rate might be slower because the buffer might have to be downloaded to the sound card, which might 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.

Remarks

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 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.

See Also

IDirectSound::CreateSoundBuffer, **IDirectSoundBuffer::GetCaps**

DSBPOSITIONNOTIFY

The **DSBPOSITIONNOTIFY** structure is used by the **IDirectSoundNotify::SetNotificationPositions** method.

```
typedef struct {
    DWORD   dwOffset;
    HANDLE  hEventNotify;
} DSBPOSITIONNOTIFY, *LPDSBPOSITIONNOTIFY;

typedef const DSBPOSITIONNOTIFY *LPCDSBPOSITIONNOTIFY;
```

Members

dwOffset

Offset from the beginning of the buffer where the notify event is to be triggered, or DSBPN_OFFSETSTOP.

hEventNotify

Handle to the event to be signaled when the offset has been reached.

Remarks

The DSBPN_OFFSETSTOP value in the **dwOffset** member causes the event to be signaled when playback or capture stops, either because the end of the buffer has been reached (and playback or capture is not looping) or because the application called the **IDirectSoundBuffer::Stop** or **IDirectSoundCaptureBuffer::Stop** method.

DSBUFFERDESC

The **DSBUFFERDESC** structure describes the necessary characteristics of a new DirectSoundBuffer object. This structure is used by the **IDirectSound::CreateSoundBuffer** method.

```
typedef struct {
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
} DSBUFFERDESC, *LPDSBUFFERDESC;

typedef const DSBUFFERDESC *LPCDSBUFFERDESC;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags

Identifies the capabilities to include when creating a new DirectSoundBuffer object. Specify one or more of the following:

DSBCAPS_CTRL3D

The buffer is either a primary buffer or a secondary buffer that uses 3-D control.

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_CTRLPOSITIONNOTIFY

The buffer must have position notification capability.

DSBCAPS_CTRLVOLUME

The buffer must have volume control capability.

DSBCAPS_GETCURRENTPOSITION2

Indicates that **IDirectSoundBuffer::GetCurrentPosition** should use the new behavior of the play cursor. In DirectSound in DirectX 1, the play cursor was significantly ahead of the actual playing sound on emulated sound cards; it was directly behind the write cursor. Now, if the DSBCAPS_GETCURRENTPOSITION2 flag is specified, the application can get a more accurate play position. If this flag is

not specified, the old behavior is preserved for compatibility. Note that this flag affects only emulated sound cards; if a DirectSound driver is present, the play cursor is accurate for DirectSound in all versions of DirectX.

DSBCAPS_GLOBALFOCUS

The buffer is a global sound buffer. With this flag set, an application using DirectSound can continue to play its buffers if the user switches focus to another application, even if the new application uses DirectSound. The one exception is if you switch focus to a DirectSound application that uses the DSSCL_EXCLUSIVE or DSSCL_WRITEPRIMARY flag for its cooperative level. In this case, the global sounds from other applications will not be audible.

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_MUTE3DATMAXDISTANCE

The sound is to be reduced to silence at the maximum distance. The buffer will stop playing when the maximum distance is exceeded, so that processor time is not wasted.

DSBCAPS_PRIMARYBUFFER

Indicates that the buffer is a primary sound buffer. If this value is not specified, a secondary sound 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 are muted, but the sticky focus buffers are still audible. This is useful for nongame applications, such as movie playback (DirectShow), when the user wants to hear the soundtrack while typing in Microsoft Word or Microsoft Excel, for example. However, if the user switches to another DirectSound application, all sound buffers, both normal and sticky focus, in the previous application are muted.

dwBufferBytes

Size of the new buffer, in bytes. This value must be 0 when creating primary buffers. For secondary buffers, the minimum and maximum sizes allowed are specified by DSBSIZE_MIN and DSBSIZE_MAX, defined in Dsound.h.

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.

Remarks

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**.

See Also

IDirectSound::CreateSoundBuffer

DSCAPS

The **DSCAPS** structure specifies the capabilities of a DirectSound device for use by the **IDirectSound::GetCaps** method.

```
typedef {
    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;

typedef const DSCAPS *LPCDSCAPS;
```

Members

dwSize

Size of the structure, in bytes. This member must be initialized before the structure is used.

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 hertz (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 sound 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 sound 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.

dwMaxHwMixingAllBuffers

Specifies the total number of buffers that can be mixed in hardware. This member can be less than the sum of **dwMaxHwMixingStaticBuffers** and **dwMaxHwMixingStreamingBuffers**. Resource tradeoffs frequently occur.

dwMaxHwMixingStaticBuffers

Specifies the maximum number of static sound buffers.

dwMaxHwMixingStreamingBuffers

Specifies the maximum number of streaming sound buffers.

dwFreeHwMixingAllBuffers, **dwFreeHwMixingStaticBuffers**, and **dwFreeHwMixingStreamingBuffers**

Description of the free, or unallocated, hardware mixing capabilities of the device. An application can use these values 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 3-D positional capabilities of the device.

dwFreeHw3DAllBuffers, **dwFreeHw3DStaticBuffers**, and **dwFreeHw3DStreamingBuffers**

Description of the free, or unallocated, hardware 3-D positional capabilities of the device.

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.

dwUnlockTransferRateHwBuffers

Description of the rate, in kilobytes per second, at which data can be transferred to hardware static sound buffers (those located in on-board 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 need not be transferred anywhere. Similarly, the play processing overhead for hardware buffers is 0 because the mixing is done by the sound device.

dwReserved1 and **dwReserved2**

Reserved for future use.

See Also

IDirectSound::GetCaps

DSCBCAPS

The **DSCBCAPS** structure is used by the **IDirectSoundCaptureBuffer::GetCaps** method.

```
typedef struct
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwBufferBytes;
    DWORD   dwReserved;
} DSCBCAPS, *LPDSCBCAPS;

typedef const DSCBCAPS *LPCDSCBCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be NULL or the following flag:

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
The size, in bytes, of the capture buffer.

dwReserved
Reserved for future use.

DSCBUFFERDESC

The **DSCBUFFERDESC** structure is used by the **IDirectSoundCapture::CreateCaptureBuffer** method.

```
typedef struct
{
    DWORD                   dwSize;
    DWORD                   dwFlags;
    DWORD                   dwBufferBytes;
    DWORD                   dwReserved;
    LPWAVEFORMATEX          lpwfxFormat;
} DSCBUFFERDESC, *LPDSCBUFFERDESC;

typedef const DSCBUFFERDESC *LPCDSCBUFFERDESC;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
Specifies device capabilities. Can be NULL or the following flag:

DSCBCAPS_WAVEMAPPED
The Win32 wave mapper will be used for formats not supported by the device.

dwBufferBytes
Size of capture buffer to create, in bytes.

dwReserved
Reserved for future use.

lpwfxFormat
Pointer to a **WAVEFORMATEX** structure containing the format in which to capture the data.

DSCCAPS

The **DSCCAPS** structure is used by the **IDirectSoundCapture::GetCaps** method.

```
typedef struct
{
    DWORD   dwSize;
    DWORD   dwFlags;
    DWORD   dwFormats;
    DWORD   dwChannels;
} DSCCAPS, *LPDSCCAPS;

typedef const DSCCAPS *LPCDSCCAPS;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

dwFlags
No flags are currently defined.

dwFormats
Standard formats that are supported. See the reference for the **WAVEINCAPS** structure in the Platform SDK.

dwChannels
Number specifying the number of channels supported by the device, where 1 is mono, 2 is stereo, and so on.

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 can return, 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_NOINTERFACE

The requested COM interface is not available.

DSERR_OTHERAPPHASPRIO

Another application has a higher priority level, preventing this call from succeeding

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.

About DirectPlay

The Microsoft® DirectPlay® application programming interface (API) for Microsoft Windows® 95 is a software interface that simplifies application access to communication services. DirectPlay has become a technology family that not only provides a way for applications to communicate with each other, independent of the underlying transport, protocol, or online service, but also provides this independence for matchmaking servers and game servers.

Applications (especially games) can be more compelling 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 the developer to deal with the differences that each connectivity solution represents, DirectPlay provides well-defined, generalized communication capabilities. DirectPlay shields developers from the underlying complexities of diverse connectivity implementations, freeing them to concentrate on producing a great application.

What's New in DirectPlay 5?

This section discusses new features in DirectPlay 5. For the most recent updates, including new features, additional samples, and further technical information, consult the Microsoft DirectX web site at http://www.microsoft.com/DirectX.

DirectPlay 5 has a new interface, IDirectPlay3. This interface inherits directly from **IDirectPlay2** and by default behaves as **IDirectPlay2**. All new functionality is enabled through new methods or new flags.

DirectPlay 5 supports the following new features and methods:

- DirectPlay interface objects can be created directly by using the **CoCreateInstance** method. This eliminates the need to link directly to the dplayx.dll.
- IDirectPlay3::EnumConnections enumerates the service providers and lobby providers available to the application. This method supersedes DirectPlayEnumerate.
- IDirectPlay3::InitializeConnection initializes a DirectPlay connection. This method supersedes DirectPlayCreate.
- The new IDirectPlayLobby2::CreateCompoundAddress method creates an address to pass to the InitializeConnection method.
- IDirectPlay3::SecureOpen creates or joins a session on a server that requires security.
- IDirectPlay3::CreateGroupInGroup, IDirectPlay3::AddGroupToGroup, IDirectPlay3::DeleteGroupFromGroup, and IDirectPlay3::EnumGroupsInGroup add richer group functionality and navigation. The new IDirectPlay3::GetGroupFlags and IDirectPlay3::GetGroupParent methods give ready access to additional group information.
- IDirectPlay3::SendChatMessage enables players to chat with other players in a session or connected to a lobby server, using standardized messages.
- IDirectPlay3::SetGroupConnectionSettings, IDirectPlay3::GetGroupConnectionSettings, and IDirectPlay3::StartSession enable synchronized application launching from a lobby server.
- The new DPCREDENTIALS structure holds the user name, password, and domain to use when connecting to a secure server. The DPSECURITYDESC structure describes the security properties of a DirectPlay session instance.
- The new DPCOMPOUNDADDRESSELEMENT structure describes a DirectPlay Address data chunk that can be used to create longer DirectPlay Addresses.
- The new IDirectPlay3::GetPlayerAccount method and new DPACCOUNTDESC can be used by a session host to obtain account information for a player logged into a secure session.
- The new IDirectPlay3::GetPlayerFlags method gives access to a player's flag settings.

DirectPlay 5 also supports new functionality for existing DirectPlay 3 methods:

- An application can create multiple DirectPlay objects.
- The IDirectPlay3::SetSessionDesc method enables the host to change the session description.
- The IDirectPlay3::EnumSessions method can now be called asynchronously and will maintain a constantly refreshing list of sessions available on the session
- Password protection of sessions has been improved. Specify the DPENUMSESSIONS_PASSWORDREQUIRED flag in EnumSessions to enumerate password-protected sessions (in addition to nonpassword-protected sessions). The DPSESSIONDESC2 structure will contain a flag indicating that the session needs a password. Put the password in the DPSESSIONDESC2 structure passed to the Open method to join the session.
- Applications can override the service provider dialog boxes that prompt users for information. To prevent these dialog boxes from appearing, create a DirectPlay Address using the IDirectPlayLobby2::CreateAddress or IDirectPlayLobby2::CreateCompoundAddress methods and then call IDirectPlay3::InitializeConnection with this DirectPlay Address. A subsequent call to IDirectPlay3::EnumSessions will not display a dialog box prompting the user for address information.
- A new multicast server option improves group messaging.
- Support has been added for scalable client/server architecture applications.
- Sessions can be hosted securely and require users to log in with a name and password.
- Members of a secure session can send digitally signed or encrypted messages, by using the

DPSEND_SIGNED and DPSEND_ENCRYPTED flags in the Send method.

Updates to DirectPlay

Be sure to consult the Microsoft DirectX web site for the latest information about new features and updates to DirectPlay, additional samples, and further technical information about using DirectPlay.

The web site is http://www.microsoft.com/DirectX.

Writing a Network Application

The fundamental concern when writing a networked, multiplayer application is how to propagate state information to all the computers in the session. The state of the session defines what users see on their computers and what actions they can perform. Generally, two things make up the session state: the environment, and the individual users or players.

The environment consists of the objects that the players can manipulate or interact with. This can include a map of a dungeon, a racecourse, or even a document. The environment can also include computer-controlled players.

Within the environment, each player also has a state indicating the player's current properties. These can include position, velocity, energy, armor, and so on.

When a player first joins the session, the session's current state must be downloaded, including the environment and the state of the other players.

Actions performed by users or the natural progression of a game change the state of the session. When this happens, the change must be propagated to the other computers in the session through messages. There are many techniques for updating the session state, but they all depend on sending messages between computers.

When two players perform an action on the same object (like opening a door) or on each other (like swinging a sword) a conflict can arise. That is, two players are trying to do something where only one of them can succeed. In these cases, arbitration or conflict resolution is needed. There are several ways to resolve conflicts like this: the two players can communicate with each other and decide which one wins, or a special player can arbitrate all conflicts.

The actions of players change the state of a game, and conflicting actions must be resolved before the game's state can be updated. There are two fundamental ways of maintaining state in a session. The first way is called peer-to-peer. In peer-to-peer sessions, all the computers in the session have the complete state of the environment and all the players. Any change in state that happens on one computer must be propagated to all the other computers in the session. No one computer is really in charge.

The second way to maintain the session state is called client/server. In client/server sessions, one computer is designated the server, and it maintains the complete state of the game and performs conflict resolution. All the other computers in the session are clients, and they download some portion of the state from the server. When they need to change the state, they tell the server, and the server propagates the change to other clients as needed.

With DirectPlay, you can create and manage both peer-to-peer and client/server applications. DirectPlay provides all the tools needed to write a networked, multiplayer application by providing the services to manage players, send messages between players, and automatically propagate the state among all the computers.

DirectPlay Overview

This section contains the following topics, which provide general information about the DirectPlay component.

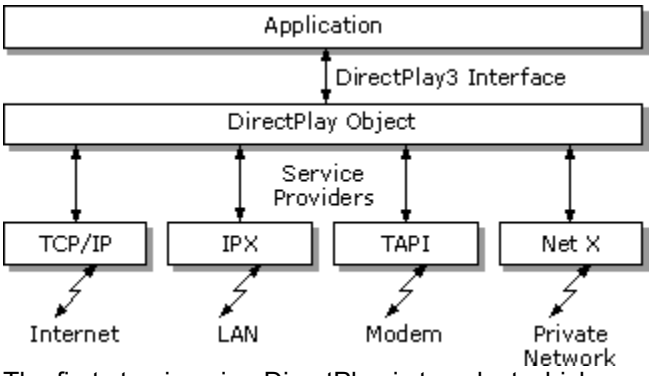
- Architecture
- Session Management
- Player Management
- Message Management
- Group Management
- Overview of DirectPlay Communications

Architecture

The DirectPlay API is a network abstraction and distributed object system that applications can be written to. The API defines the functionality of the abstract DirectPlay network and all the functionality is available to your application regardless of whether the actual underlying network supports it or not. In cases where the underlying network does not support a method, DirectPlay contains all the code necessary to emulate it. Examples include group messaging and guaranteed messaging.

DirectPlay's service provider architecture insulates the application from the underlying network it is running on. The application can query DirectPlay for specific capabilities of the underlying network, such as latency and bandwidth, and adjust its communications accordingly.

The following diagram illustrates the DirectPlay service provider architecture.



The first step in using DirectPlay is to select which service provider to use. The service provider determines what type of network or protocol will be used for communications. The protocol can range from TCP/IP over the Internet to an IPX local area network to a serial cable connection between two computers.

Use the service provider to make a connection to a point on the network. The user may need to provide additional information to make a connection, or the application can specify the connection parameters.

Connection Management Methods

DirectPlay provides two very useful connection management methods:

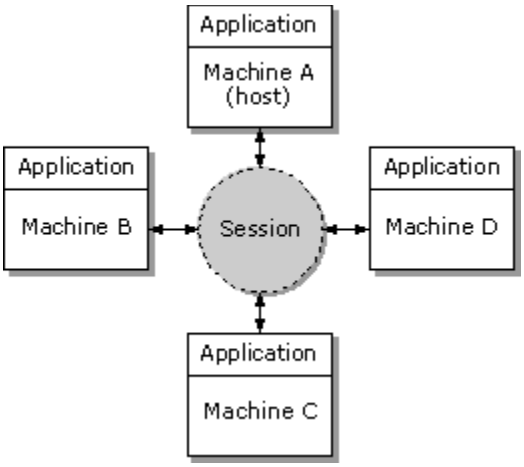
- IDirectPlay3::EnumConnections enumerates all the connections that are available to the application.
- IDirectPlay3::InitializeConnection initializes a specific connection.

Session Management

A DirectPlay session is a communications channel between several computers. Before an application can start communicating with other computers it must be part of a session. An application can do this in one of two ways: it can enumerate all the existing sessions on a network and join one of them, or it can create a new session and wait for other computers to join it. Once the application is part of a session, it can create a player and exchange messages with all the other players in the session.

Each session has one computer that is designated as the host. The host is the owner of the session and is the only computer that can change the session's properties.

The following diagram illustrates the DirectPlay session model.



Session Management Methods

DirectPlay provides several session management methods:

- IDirectPlay3::EnumSessions enumerates all the available sessions.
- IDirectPlay3::Open joins one of the enumerated sessions or creates a new session.
- IDirectPlay3::SecureOpen performs the same function as IDirectPlay3::Open, but enables the application to alter the default opening behavior.
- IDirectPlay3::Close leaves the currently open session.
- IDirectPlay3::GetSessionDesc gets the properties of the current session.
- IDirectPlay3::SetSessionDesc changes the properties of the current session.
- IDirectPlay3::GetCaps gets the communications capabilities of the underlying network.

Player Management

The most basic entity within a DirectPlay session is a player. A player represents a logical object within the session that can send and receive messages. DirectPlay does not have any representation of a physical computer in the session. Each player is identified as being either a local player (one that exists on your computer) or a remote player (one that exists on another computer). Each computer must have at least one local player before it can start sending and receiving messages. Individual computers can have more than one local player.

When an application sends a message, it is always directed to another player — not another computer. The destination player can be another local player (in which case the message will not go out over the network) or a remote player. Similarly, when an application receives messages they are always addressed to a specific (local) player and marked as being from some other player (except system messages, which are always marked as being from DPID_SYSMSG).

DirectPlay provides some additional player management methods that an application can use. These methods can save the application from having to implement a list of players and data associated with each one. You do not need these methods to use DirectPlay successfully. They enable an application to associate a name with a player and automatically propagate that name to all the computers in the session. Similarly, the application can associate some arbitrary data with a player that will be propagated to all the other computers in the session. The application can also associate private local data with a player that is available only to the local computer.

Basic Player Management Methods

DirectPlay provides several basic player management methods:

- IDirectPlay3::EnumPlayers enumerates all the players in the sessions.
- IDirectPlay3::CreatePlayer creates a local player.
- IDirectPlay3::DestroyPlayer destroys a local player.

Additional Player Management Methods

DirectPlay provides these additional methods to manage player information:

- IDirectPlay3::GetPlayerCaps gets a player's communications capabilities.
- IDirectPlay3::GetPlayerName gets a player's name.
- IDirectPlay3::SetPlayerName changes a player's name.
- IDirectPlay3::GetPlayerData gets the application-specific data associated with a player.
- IDirectPlay3::SetPlayerData changes the application-specific data associated with a player.
- IDirectPlay3::GetPlayerAddress gets a player's network-specific address.
- IDirectPlay3::GetPlayerAccount gets a player's account information.
- IDirectPlay3::GetPlayerFlags gets a player's flag settings.

Message Management

Once an application has created a player within a session, it can start exchanging messages with other players in the session. DirectPlay imposes no message format or extra bytes on the message. A message can be sent to an individual player, to all the players in the sessions, or to a subset of players that have been defined as a group (see Group Management). When sending a message it must also be marked as being from a specific local player.

All the messages received by an application are put into a receive queue. The application must retrieve individual messages from the queue and act on them as appropriate. The application can either poll the receive queue for messages or use a separate thread that waits on a synchronization event for notification that a new message has arrived.

There are two types of messages. Player messages are messages that another player in the session sent. This type of message is directed to a specific player and is marked as being from the sending player. System messages are sent to all the players in a session and are all marked as being from the system (DPID_SYSMSG). DirectPlay generates system messages to notify the application of some change in state of the session; for example, when a new player has been created. See Using System Messages for more information.

Basic Message Management Methods

DirectPlay provides several message management methods:

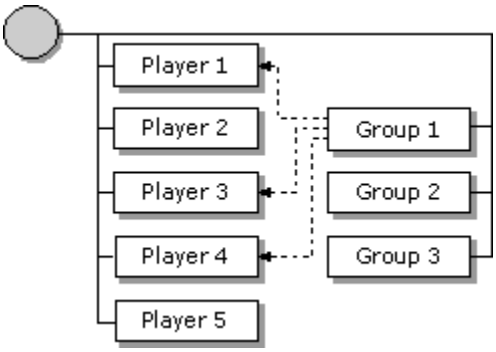
- IDirectPlay3::Send sends a message from a local player to another player in the session.
- IDirectPlay3::SendChatMessage enables players to chat with other players using standardized messages.
- IDirectPlay3::Receive retrieves a message from the incoming message queue.
- IDirectPlay3::GetMessageCount gets the number of messages currently in the incoming message queue.

Group Management

DirectPlay supports groups within a session. A group is logical collection of players. By creating a group of players, an application can send a single message to the group and all the players in the group will receive the message. A group is the means by which a network's multicast capabilities are exposed to the application.

Groups can also be used as a general means to organize players in a session. A player can belong to more than one group. DirectPlay provides methods for administering groups and their membership. Additional methods associate names and data with individual groups as a convenience, but you don't need them to use groups.

The following diagram shows a logical representation of the contents of a DirectPlay session.



Basic Group Management Methods

DirectPlay provides several group management methods:

- IDirectPlay3::EnumGroups enumerates all the groups in the session.
- IDirectPlay3::CreateGroup creates a new group.
- IDirectPlay3::DestroyGroup destroys a group that this computer created.
- IDirectPlay3::EnumGroupPlayers enumerates the players that are in a group.
- IDirectPlay3::AddPlayerToGroup adds a player to a group.
- IDirectPlay3::DeletePlayerFromGroup removes a player from a group.

Additional Group Management Methods

DirectPlay provides these additional methods to manage group information:

- IDirectPlay3::GetGroupName gets the group's name.
- IDirectPlay3::SetGroupName changes the name of a group created by this computer.
- IDirectPlay3::GetGroupData gets the application-specific data associated with a group.
- IDirectPlay3::SetGroupData changes the application-specific data associated with a group created by this computer.
- IDirectPlay3::GetGroupFlags gets the flags describing a group.
- IDirectPlay3::GetGroupParent gets the DPID of the parent of the group.

New Group Management Methods

DirectPlay 5 has added these methods that manage groups within groups and shortcuts to groups:

- IDirectPlay3::CreateGroupInGroup creates a group within an existing group.
- IDirectPlay3::EnumGroupsInGroup enumerates all the groups within another group.
- IDirectPlay3::AddGroupToGroup adds a shortcut from a group to an already existing group.
- IDirectPlay3::DeleteGroupFromGroup removes a group previously added to another with AddGroupToGroup, but doesn't destroy the deleted group.
- IDirectPlay3::GetGroupConnectionSettings retrieves a group's connection settings.
- IDirectPlay3::SetGroupConnectionSettings sets a group's connection settings.

Overview of DirectPlay Communications

DirectPlay's default mode of communications is peer-to-peer. In this model, the session's complete state is replicated on all the computers in the session. This means that the session description data, the list of players and groups, and the names and remote data associated with each session are duplicated on every computer. When one computer changes something, it is immediately propagated to all the other computers.

DirectPlay's alternative mode of communications is client/server. In this model, only the server stores the session's complete state and each client has only a subset of the session's state. Each client has only the information that is relevant to that computer and receives that information from the server. When one computer changes something, it propagates the change to the server. The server then determines which clients it must inform of the change.

An application can manage its own data using either a client/server model or peer-to-peer model, but this will not change how the underlying DirectPlay session state is managed.

The following sections discuss the two modes of communications within DirectPlay sessions:

- Peer-to-Peer Session
- Client/Server Session

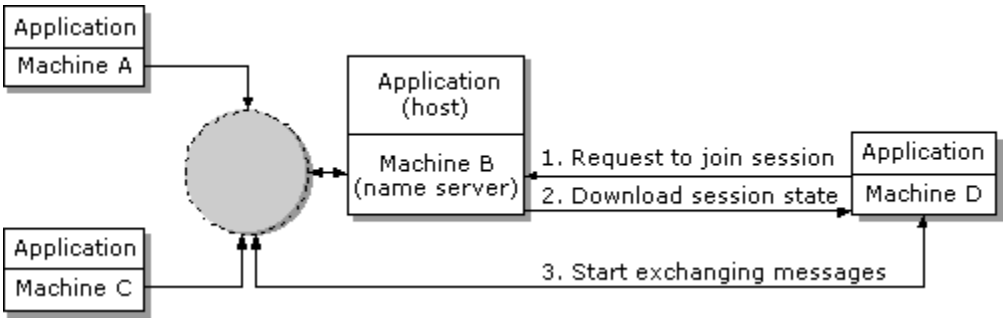
Peer-to-Peer Session

In a peer-to-peer DirectPlay session, one computer is designated the name server. This computer responds to enumeration requests, regulates computers trying to join the session, downloads the session's state to new computers that have joined, and generates ID numbers as players and groups are created. Beyond that, the name server is just another peer in the session and runs the same application as all the other peers. Messages are not routed through the name server and the name server does not generate all the system messages.

DirectPlay automatically performs the name server functions, which are not exposed to the application in any way. The DirectPlay application running on the name server is also the session's host. Only the host application can change the session description data. Each peer application in the session has access to the complete list of players and groups in the session (see Session Management) and can send and receive messages from any other player in the session or send a message to any group in the session. See Session Management for more details.

The session's network address corresponds to the name server's network address. When a computer needs to join the session, it sends the join request to the name server. In response, the name server downloads the session's state to new computers. If the name server leaves the session, a new computer is elected name server. When the name server migrates, the session's network address also changes.

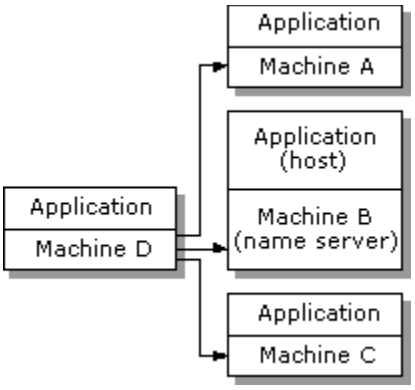
The following diagram illustrates joining a session. To join a session, a machine sends the join request to the name server of the session.



Peer-to-peer sessions generally have a limitation on the number of computers that can participate. Every change in state on every computer (like player movements) must be broadcast to all the other computers in the session. Because there is a limit on how much data a computer can receive (especially if connected through a phone line) there is a limit to how many computers can generate data. Minimizing the quantity and frequency of data exchanged can help increase the number of computers that can be in a session before performance degrades.

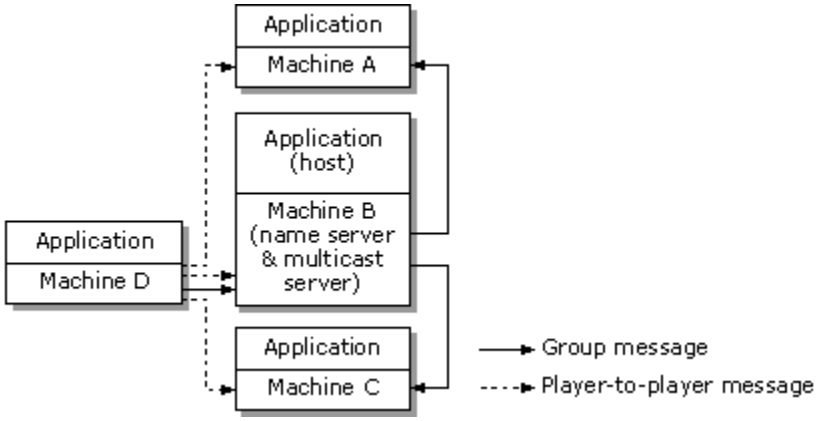
Computers in a peer-to-peer DirectPlay session communicate directly with each other. That is, messages are not sent through an intermediate computer to reach their destination (although they can go through a router). When DirectPlay needs to send the same message to a set of computers, it attempts to use any multicast capabilities in the service provider. If multicast is not supported, then individual unicast messages are sent to each destination computer.

The following diagram illustrates group messaging without multicast. To broadcast a message, individual messages are sent to each machine.



This can be a source of inefficiency in sessions with a large number of players (more than four). Adding a multicast server to the session can alleviate this. A multicast server is a computer that will forward a single message to multiple destinations. A computer needing to broadcast a message to all the other computers in the session can send one message to the multicast server, which in turn sends individual messages to all the other computers. This is more efficient, because the multicast server is connected to the network through a high-speed link and can therefore pump out unicast messages faster than any individual computer. The assumption is that the server has a high-speed link to the network (such as a T1 line) while the other computers have slow-speed links (such as phone lines).

The following diagram illustrates group messaging using a multicast server. Player-to-player messages are sent directly to the destination machine. Group messages are sent to the multicast server, which forwards them to all the destination machines.



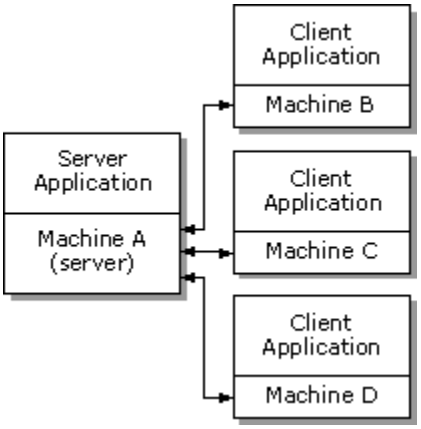
DirectPlay supports the creation of a multicast server in a session. If the name server is on a sufficiently high-speed link, it can choose to be a multicast server for the session as well. Adding a multicast server to a session only changes how group messages (including broadcast) are routed — the application's behavior doesn't change. Unicast player-to-player messages are still sent directly to the destination computers.

Client/Server Session

In a DirectPlay client/server session, one computer is designated the server. Like the peer-to-peer name server, this computer responds to enumeration requests, regulates computers trying to join the session, downloads the session's state to new computers that have joined, and generates ID numbers as players and groups are created. Unlike the peer-to-peer name server, all messages in the session are routed through the server.

The server computer runs a special version of the application (the application server) that can maintain the master state of the game, updating that state according to actions that the client computers take, and notifying individual clients of relevant events. The application server creates a special player called the server player. Any client computer can send a message directly to the server player or receive a message from the server player. When the server player receives a message it can send messages to other client players to inform them of a change in state.

The following diagram illustrates client\server communications. Each machine communicates directly with the server only. The server can forward messages to other clients.

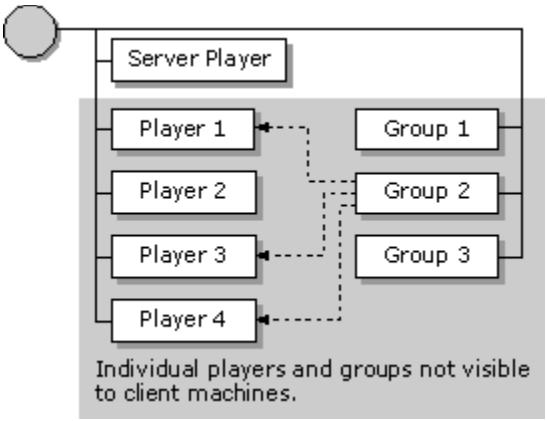


The client computers run client versions of the application that can maintain the local state, updating the state in response to messages from the server, and sending messages to the server when some local action has occurred.

You can implement a client/server application in two ways. One mode is a hybrid peer-to-peer and client/server session, where every client application has the complete player and group list available to it (including the server player). Any client player can send a message directly to any player in the session, and each player or group can have data associated with it (such as name or remote data) that will be updated on all the computers whenever it changes. All the benefits of peer-to-peer state propagation are available, and, in addition, a server player is available. However, you can't control the propagation of the player-to-player data. Depending on the frequency with which players and groups are created and destroyed or their data changes, client computers could be overloaded with in-coming messages. This type of a session is not scalable beyond about 16 players. It isn't very different from a peer-to-peer session, except that it has a server player, all messages are routed through the server, and the server player will receive a copy of all the messages that pass through the server.

The second mode is pure client/server. Here, each client application sees only the server player and its own local players in the session. A client player can send messages only to the server player and receive messages from only the server player. The application server can see all the players on all clients. When the server receives a message from a client, it can intelligently decide how to update the master application state and which clients it must inform of this update. In this mode, the client application must manage the player list and the data associated with each player.

The following diagram shows a logical representation of the contents of a DirectPlay client/server session.



Like a peer-to-peer session, the session's network address corresponds to the server's network address. When a computer needs to join the session, it sends the join request to the server. In response, the server downloads the session's state to the new computers. If the server leaves the session, the session is terminated.

Because all messages are routed through the server, it automatically behaves as a multicast server.

Security

Using DirectPlay security features, an application running on a server can create secure sessions. DirectPlay implements security through the Security Support Provider Interface (SSPI) on Windows. Key features supported by DirectPlay in a secure session are:

- Authentication to verify the identity of the user. Once a user has been authenticated, communications between the client and server can be done securely either by digitally signing or encrypting the messages.
- Digitally signed system messages to verify the identity of the sender.
- Encryption of sensitive system messages.

For more information, including how to set up a secure server, start a secure session, and specify security packages, see Security and Authentication.

DirectPlay Lobby Overview

A DirectPlay lobby server is a common place on a network that (at a minimum) tracks DirectPlay application sessions in progress, and users that are connected to the server. Users can navigate around the lobby server to find areas of interest. At any location on the lobby server the user can chat with other users, join sessions that are in progress, or gather a group of players to start a new session.

A lobby server's main advantage is that it acts as a central, well-known location where users can go to find sessions and other people to interact with. The lobby server manages all the network addresses of the various players and sessions and can automatically manage launching applications and connecting them to the correct address without user intervention. The management of sessions, players, and their network addresses is especially useful on the Internet, where users generally can't find opponents or get applications connected in a session easily. It also has the advantage of launching application sessions for the user without the user having to enter any network configuration information.

Lobby servers can be greatly enhanced to provide more services to the user, such as tournaments, individual score tracking and high scores, personal profiles, avatars, message boards, news, a broader user interface, authenticated membership, software updates, and so on.

A lobby-aware application is one that has been specifically developed to interoperate with lobby servers. There are two types of lobbies that a user can experience. An external lobby is a client application whose sole purpose is to interact with a lobby server and the other users connected to it. When the time comes to start or join an application session, the lobby client launches the application in a separate process and gives it all the information necessary to establish a connection to the session. Adding support for external lobby launching to a DirectPlay application is quite straightforward and explained further in Supporting External Lobby Launching.

An internal lobby is a lobby user interface that is integrated into the application itself. This is more difficult to implement because the application must implement the user interface for the lobby as well as that for the game. However, it has the advantage that you can customize the lobby experience to match the application's theme.

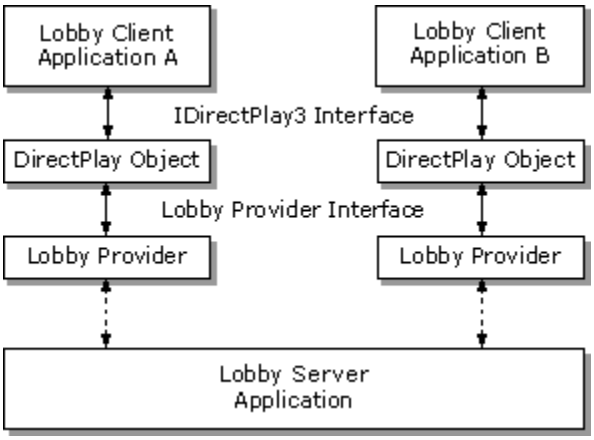
This section discusses the following topics.

- DirectPlay Lobby Architecture
- Lobby Sessions
- Lobby Navigation
- Synchronized Launching

DirectPlay Lobby Architecture

The DirectPlay lobby architecture consists of a client API that all applications, whether they are external lobby clients or applications that have a lobby client built in, use to connect to any DirectPlay-compliant lobby server. This is done through a lobby provider interface that, like a service provider, abstracts the interaction with the lobby server. The author of the lobby server application must write the lobby provider that resides on the client computer. The application calls the standard DirectPlay APIs, and the lobby provider's dynamic-link library (DLL) services these methods by communicating with the lobby server software.

The following diagram illustrates the DirectPlay lobby architecture. Different lobby client applications can connect to the same lobby server through the DirectPlay API.



At the very least, the lobby server must be able to track all the users currently connected to it, organize those players by grouping them, and synchronize the launch of an application session.
The DirectPlay API defines a common level of functionality for all lobby servers. Any generic lobby client application can connect to any generic lobby server and, through the DirectPlay API and the lobby provider architecture, they can interoperate successfully. An application can extend the basic functionality with Send and Receive. A lobby client designed to work with a specific lobby server can implement extended functionality that is not defined by the DirectPlay API.

There is no separate API to communicate with a DirectPlay lobby server. Interaction with a lobby server has been abstracted so that the same IDirectPlay3 interface used to communicate in an application session can be used to communicate with the lobby server. Additional methods and messages have been added to the interface to support the additional functionality that a lobby server requires.

By using the same DirectPlay methods to interact with the lobby server, it is simple to add a lobby client interface to an application, because the same APIs and concepts are being leveraged.

Lobby Sessions

A lobby session closely resembles a DirectPlay client/server session (see the illustration in Client/Server Session). The term *lobby session* refers to a connection to a lobby server where clients and the server have not been specifically written to interoperate. The term *application session* refers to a traditional DirectPlay session in which all the clients and the server (if any) have been specifically written to interoperate.

A DirectPlay lobby session is used to represent a connection to a lobby server. Like a DirectPlay application, the first step in using DirectPlay to communicate with a lobby server is to select which lobby provider to use and which lobby server to connect to. Like a DirectPlay application, the IDirectPlay3::EnumConnections and IDirectPlay3::InitializeConnection methods are used to do this.

The lobby client uses the same session management methods as an application client to locate and to join a lobby session; for example, EnumSessions, Open, SecureOpen, and Close. Joining a lobby session gives the client application access to all the information on the lobby server and enables the user to interact with other users on the lobby server.

Like a DirectPlay application session, the player is the basic entity in a lobby session. Each player represents a user on a client computer connected to the lobby server. There is also the server player representing the lobby server. Once a lobby client connects to a lobby server (by joining the lobby session), it must establish the user's presence by creating a player and adding the player to an initial group before the player can start communicating with the server and other players. In fact, other users connected to the session won't even be aware of the new user's presence until this happens.

The same player management methods used by application sessions are used to manage players in a lobby session; for example, EnumPlayers, CreatePlayer, DestroyPlayer, GetPlayerName, SetPlayerName, and GetPlayerCaps.

Unlike a DirectPlay application session, Send and Receive generally cannot be used to exchange messages with other players. In the DirectPlay lobby architecture, any application that uses the DirectPlay API can connect to any lobby server. This means that many different lobby client applications might be present in the same lobby session, all of which were written by different developers. Simply sending a message to another client player or the server player does not guarantee that it will be interpreted correctly by the recipient application.

Common functions that require communications between clients and the server have new methods created for them. DirectPlay determines the message's format. The recipient receives a system message containing the content of the message in a well-defined data structure. For example, the IDirectPlay3::SendChatMessage method sends a text chat message to another player.

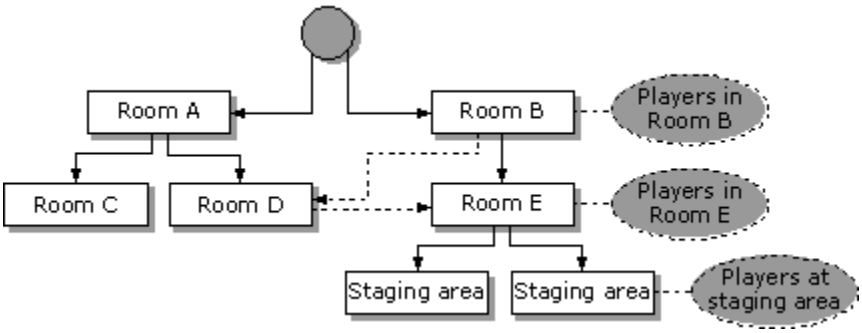
Groups take on more significance in a lobby session, because they are used to define the organization of the entire lobby server and for launching application sessions. You can create a complete hierarchy of groups to manage and organize all the players and sessions that the lobby server tracks.

For the purposes of the lobby server, groups are classified as one of two types. A standard group can contain players and other groups and is called a **room**. A room is primarily used as a meeting place for players to interact with other players in the context of the lobby. The room contains links to other rooms and links to the second type of group — a **staging area**.

A staging area group typically contains only players. A staging area is used to marshal players together in order to launch a new session. Once the session has been launched, the staging area can remain in existence so that new players can join the session in progress.

Room groups are primarily organized in a hierarchical structure. A few top-level groups are created and other groups are created within existing groups. The lobby client can enumerate all the top-level groups as well as the groups contained within a group. The lobby server can also organize the groups in a web structure. Once a group is created, you can add it to another group, which creates a link between the two groups. When a lobby client enumerates groups within a group, all subgroups and linked groups are returned.

The following diagram illustrates the organization of a lobby session.



Players can only be seen in groups. They cannot be seen until they are part of a group. Players can belong to more than one group at once.

Lobby Navigation

Players can navigate through the lobby server space simply by deleting themselves from their current group and adding themselves to a new group. At any time, the IDirectPlay3::EnumGroupsInGroup method can be called to determine what groups are linked to the current group.

In general, the scope of the lobby session visible to a player is limited by the groups the player is part of. This is done to limit the amount of information that the server must download to the client.

Synchronized Launching

Application sessions are marshaled and launched from a staging area. One player creates a staging area and waits for other players to join it. A player can set the properties of the application session to be created through the IDirectPlay3::SetGroupConnectionSettings method. Other players who are considering joining the group can find out what the application session properties are by calling the IDirectPlay3::GetGroupConnectionSettings method. Once enough players have joined the staging area, any player can call the IDirectPlay3::StartSession method to initiate the synchronized launch sequence. Every player in the staging area groups receives a DPMSG_STARTSESSION system message with a DPLCONNECTION structure. If the player is using an external lobby client, it can launch the application using the IDirectPlayLobby2::RunApplication method. If the player is using an internal lobby, the application can use the information to establish a connection to the appropriate session by calling IDirectPlayLobby2::Connect.

DirectPlay Providers

This section describes the DirectPlay interface and the DirectPlay service providers and how they interact with each other. See:

- Service Providers
- Lobby Providers

The DirectPlay interface is a common interface to the application for simple send and receive type messaging as well as higher-level services like player management, groups for multicast, data management and propagation, and lobby services for locating other players on a network.

The service providers furnish network-specific communications services as requested by DirectPlay. Online services and network operators can supply service providers to use specialized hardware, protocols, communications media, and network resources. A service provider can simply be a layer between DirectPlay and an existing transport like Winsock, or it can use specialized resources on an online service such as multicast servers, enhanced quality of service, or billing services. Microsoft includes four generic service providers with DirectPlay: head-to-head modem (TAPI), serial connection, Internet TCP/IP (using Winsock), and IPX (also using Winsock).

The DirectPlay interface hides the complexities and unique tasks required to establish an arbitrary communications link inside the DirectPlay provider implementation. An application using DirectPlay need only concern itself with the performance and capabilities of the virtual network presented by DirectPlay. It need not know whether a modem, network card, or online service is providing the medium.

DirectPlay will dynamically bind to any DirectPlay provider installed on the user's system. The application 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, protocol, and other network resources.

The DirectPlay API is exposed to the application through several COM interfaces. (See DirectPlay Interfaces for a discussion of COM and DirectPlay.) Application sessions can talk to someone else on the network through the IDirectPlay3 and IDirectPlay3A interfaces. The first uses Unicode strings in all the DirectPlay structures, while the second uses ANSI strings. **IDirectPlay**, **IDirectPlay2**, and **IDirectPlay2A** still exist for backward compatibility with applications written to a previous version of the DirectPlay SDK.

Lobby clients can talk to another application on the same machine through the IDirectPlayLobby2 and **IDirectPlayLobby2A** interfaces. The first interface uses Unicode strings while the second uses ANSI strings. The **IDirectPlayLobby** interface still exists for backward compatibility.

Service Providers

The service provider furnishes network-specific communication services as requested by DirectPlay. Online services and network operators can supply service providers for specialized hardware and communications media. Microsoft includes the following service providers with DirectPlay:

- TCP/IP
- IPX
- Modem-to-Modem
- Serial Link

Individual service providers are identified using a GUID. GUIDs for the standard service providers are listed in the header file DPLAY.H. Third-party service providers will have their own GUIDs.

This section outlines what to expect from the default service providers that Microsoft ships. To obtain information about the behavior of third-party service providers, contact the network operator.

TCP/IP

The TCP/IP service provider uses Winsock to communicate over the Internet or local area network (LAN) using the TCP/IP protocol. It uses UDP (User Datagram Protocol) packets for nonguaranteed messaging and TCP for guaranteed messaging. A single computer can host multiple DirectPlay sessions using TCP/IP.

When the IDirectPlay3::EnumSessions method is called, TCP/IP displays a dialog box asking the user for the session's IP address. The user must enter the IP address of the computer hosting the session to be joined. If the computer has a name (such as microsoft.com), the name can be used instead of the IP address, and DirectPlay will use Domain Name System (DNS) lookup to find it. The IDirectPlay3::EnumSessions method will return the sessions that the computer is hosting. The user can also leave the address blank and hit OK. In this case, DirectPlay will broadcast a message looking for sessions. This will generally only work on a LAN and only on the same subnet.

Note: A Windows 95 user can determine his or her IP address by choosing Run from the Start menu and typing WINIPCFG as the program to run. A Windows NT user can determine his or her IP address by running IPConfig from the command line. If the user is connected to both a LAN and a dial-up Internet service provider (ISP), the computer can have two IP addresses and the correct one must be selected. Most dial-up ISPs assign a dynamic IP address that changes each time the user logs on.

An application can call IDirectPlay3::InitializeConnection, or can call IDirectPlayLobby2::SetConnectionSettings followed by a call to Connect, to supply an IP address to the service provider in a DirectPlay Address. The address must be a null-terminated ANSI or Unicode string (each has a different data type GUID). If a broadcast enumeration of sessions is desired, the address must be a zero-length string; that is, a string consisting of just the null terminator.

The DirectPlay TCP/IP service provider does not generally work through firewalls.

Adding DirectPlay lobby support can eliminate the need for users to enter an IP address if they start the game from a lobby server.

This service provider can be identified using the symbol definition DPSPGUID_TCPIP.

Note: A Windows 95 user can configure his or her computer connections to display or not display a dialog box requesting connection information when DirectPlay tries to initiate a TCP/IP connection. To suppress the display of this dialog box, follow these steps:

1. Open Control Panel.
2. Double-click the Internet icon.
3. Choose the Connection tab.
4. Clear the checkbox next to **Connect to the Internet as needed**.

IPX

The IPX service provider uses Winsock to communicate over a local area network (LAN) using the Internet Packet Exchange (IPX) protocol. The service provider only supports nonguaranteed messaging. A single computer can host only one DirectPlay session using IPX.

IPX always uses a broadcast to find sessions on the network, so the IDirectPlay3::EnumSessions method will not display a dialog box requesting IP addresses.

IPX will not enumerate sessions on another subnet.

Once a session is established, packets are sent directly between computers (they are not broadcast).

This service provider can be identified using the symbol definition DPSPGUID_IPX.

Modem-to-Modem

Modem-to-modem communication uses TAPI (Telephony Application Programming Interface) to communicate with another modem.

Creating a session (by using the IDirectPlay3::Open method) causes a dialog box to appear, asking the user which modem to wait for a call on. The IDirectPlay3::EnumSessions method will also display a dialog box asking the user what phone number to call and which modem to use. Once the information is entered, DirectPlay will dial the modem and try to find sessions hosted by the computer on the other end. In both cases, dialogs are displayed to show the progress.

The current list of available modems can be obtained from the service provider by initializing it and calling IDirectPlay3::GetPlayerAddress with a player ID of zero. The DirectPlay Address returned will contain a data chunk with the ANSI modem names (DPAID_Modem) and the Unicode modem names (DPAID_ModemW). The list of modems is a series of NULL-terminated strings with a zero-length string at the end of the list.

If you insert too many delays into the message processing, you may lose packets; for example, if you print a lot of debug information.

This service provider can be identified using the symbol definition DPSPGUID_MODEM.

Serial Link

A serial link is used to communicate with another computer through the serial ports.

Creating a session (using the IDirectPlay3::Open method) causes a dialog box to appear asking the user to configure the serial port. The IDirectPlay3::EnumSessions method will also display a dialog box asking the user to configure the serial port. You must configure the serial port the same way on both computers.

If you insert too many delays into the message processing, you may lose packets; for example, if you print a lot of debug information.

This service provider can be identified using the symbol definition DPSPGUID_SERIAL.

Lobby Providers

The lobby provider is a client component (DLL) supplied by the developer of a lobby server. It implements communications functions with the lobby server as requested by DirectPlay. A lobby client written using the DirectPlay API can interoperate with any lobby server for which a lobby provider DLL is present on the system.

The DirectX SDK installs a lobby provider for use with the test lobby server (LSERVER.EXE) included with the SDK. This lobby provider is used by the BELLHOP sample lobby client and together they can be used to test the lobby-aware functions your applications.

Using DirectPlay

This section contains the following topics that explain how to use different aspects of DirectPlay.

- Debug versus Retail DLLs
- Working with GUIDs
- DirectPlay Interfaces
- Using Callback Functions
- Building Lobby-Aware Applications
- DirectPlay Messages
- DirectPlay Address (Optional)
- Migrating from Previous Versions of DirectPlay
- DirectPlay Tools and Samples
- Security and Authentication

Debug versus Retail DLLs

The SDK has the option to install debug or retail builds of the DirectPlay DLLs. When developing software, it best to install the debug versions of the DLLs. The debug DLLs have addition code in them which will validate internal data structures and output debug error messages (using the Win32 **OutputDebugString** API) while your program is executing. When an error occurs, the debug output will give you a more detailed description of what the problem is. The debug DLLs will execute more slowly than the retail DLLs but are much more useful for debugging an application. Be sure to ship the retail version with your application.

In order to see the debug messages, it is necessary to configure your computer so that debug output will be displayed in a window or on a remote computer. An interactive development environment like Microsoft Visual C++ allows you to do this. Consult the environment's documentation for exactly how to set this up.

Working with GUIDs

Globally unique identifiers are 16-byte data structures that you can use to identify objects in a globally unique way. Whenever a GUID is required in an API, a symbol representing that GUID should be used. The symbols are either defined in one of the DirectPlay header files or the application developer must generate them. You can generate GUIDs by using the Guidgen.exe utility that comes with Microsoft Visual C++. For example, every application must define an application GUID that identifies the application that is running in a session.

Note If there are different versions of an application that cannot interoperate in the same session, they should have different application GUIDs to distinguish them.

To use DirectPlay-defined GUIDs successfully in your application, you must either define the INITGUID symbol prior to all other include and define statements, or you must link to the Dxguid.lib library. If you define INITGUID, you should define it in only one of your source modules.

DirectPlay Interfaces

All the functionality in DirectPlay is accessed through member functions on COM (Component Object Model) interfaces. To use them, an application must obtain the appropriate COM interface.

The standard method of obtaining COM interfaces is to use the Win32 **CoCreateInstance** API. To use it successfully, the application must first call the Win32 **CoInitialize** API to initialize COM and then call **CoCreateInstance**, specifying the GUID of the desired interface. For example, use the following code to obtain an IDirectPlay3A interface:

```
// C++ example
hr = CoCreateInstance( CLSID_DirectPlay, NULL, CLSCTX_INPROC_SERVER,
                       IID_IDirectPlay3A, (LPVOID*)&lpDirectPlay3A);
// C example
hr = CoCreateInstance( &CLSID_DirectPlay, NULL, CLSCTX_INPROC_SERVER,
                       &IID_IDirectPlay3A, (LPVOID*)&lpDirectPlay3A);
```

When the program is finished, all the COM interfaces must be freed by calling the Release method on each interface. Finally, the Win32 **CoUninitialize** method should be called to uninitialize COM.

If you call **CoCreateInstance** without first calling **CoInitialize** you will get a CO_E_NOTINITIALIZED error, with the error text "CoInitialize has not been called."

DirectPlay has several COM interfaces. Each interface represents a revision of an earlier version of DirectPlay in which new methods are added. COM interfaces are numbered sequentially with each revision. The latest COM interface will have all the latest functionality of DirectPlay. To access the new functionality (for example, IDirectPlay3::SendChatMessage), you must use the latest COM interface. Source code written for a earlier COM interface will work fine.

Once a COM interface is obtained, an alternate interface can be used on the same object by calling the QueryInterface method on the interface. For example, DirectPlayCreate will create a DirectPlay object and return an **IDirectPlay** interface. If your application requires an IDirectPlay3 interface, it can call QueryInterface on the **IDirectPlay** interface. Be sure to release the original **IDirectPlay** interface if it is no longer needed.

Using Callback Functions

The enumeration methods in DirectPlay are used to return a list of items to the application. The application calls an enumeration method (such as IDirectPlay3::EnumPlayers) and supplies a pointer to a callback function that it has implemented. DirectPlay will call the callback function once for each item in the list. The enumeration method will not return until all the items in the list have been returned to the application through the callback function.

It is extremely important that all callbacks be declared correctly. For example:

```
BOOL FAR PASCAL EnumConnectionsCallback(LPCGUID lpguidSP, LPVOID
lpConnection, DWORD dwConnectionSize,
LPCDPNAME lpName, DWORD dwFlags, LPVOID pContext);
```

The FAR PASCAL symbol will define the function as **_stdcall**. That means it will clean up the stack before returning to DirectPlay. Do not cast function pointers when passing them to a DirectPlay enumeration method. If there is a compiler warning about the function pointer, fix the function declaration.

Building Lobby-Aware Applications

A lobby-aware application is one that, at a minimum, supports being launched from a lobby. A matchmaking lobby is a site on the Internet where end users can find other people to play games with. Once a group of people has decided to start an application session, the lobby software can launch the application on each person's computer and have them all connect to a session. The main benefit for end users is the ease with which they can establish a session with other players. Not only does it allow a user to easily find opponents, but there is also no need for the user to:

- Select a service provider. The lobby will specify which service provider to use.
- Decide whether to host or join a session. The lobby will specify whether to create or join a session.
- Enter a network address or configure the network. The lobby will supply this information if it is needed.
- Enter the name of the player. The lobby will pass in the same name that the user connected to the lobby with.

Other benefits of the lobby are:

- It can keep track of sessions in progress and enable users to join them.
- It can receive status messages from the session and display the progress to other users on a scoreboard.
- It can obtain final scores and maintain player rankings for tournament play.

For a DirectPlay application to be lobby-aware, at a minimum it must support being launched from a lobby. You can add other features to make it integrate better with the lobby. For additional information see the following topics within this section.

- Registering Lobby-Aware Applications
- Supporting External Lobby Launching
- LobbyMessaging (Optional)

Registering Lobby-Aware Applications

To enable a lobby to launch a DirectPlay application, the application must add the following entries to the system registry. Once an application has been registered, it will be recognized as a lobby-aware application and lobbies can launch it. These registry entries must be deleted when the application is removed.

The following example shows the registry entries for the DPCHAT sample application included in the SDK.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectPlay\Applications\DPChat]
"Guid"="{5BFDB060-06A4-11D0-9C4F-00A0C905425E}"
"File"="dpchat.exe"
"CommandLine"=""
"Path"="C:\DXSDK\sdk\bin"
"CurrentDirectory"="C:\DXSDK\sdk\bin"
```

The keys and values are:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectPlay\Applications\

The key's name will be the name of the application that the IDirectPlayLobby2::EnumLocalApplications method returns. It is **DPChat** in the preceding example.

Guid

This is the 16-byte GUID that identifies the application. It is formatted as shown in the example. This should be the same GUID that is put in the **guidApplication** member of the DPSESSIONDESC2 structure when creating a session. A globally unique identifier (GUID) can be generated using the Guidgen.exe utility.

File

This is the file name of the application executable.

CommandLine

This lists any command-line parameters that are to be specified when a lobby launches the application.

Path

This is the directory that the application executable resides in.

CurrentDirectory

This is the directory to set as the current directory after launching the application executable.

Supporting External Lobby Launching

Once the application has been registered, the DirectPlay application must be able to recognize whether a lobby launched it or not. If a lobby launched it, it must follow a slightly different code path to set up the network connection. Consult the DUEL (Lobby.c) and DPCHAT (Lobby.cpp) samples in the SDK for the code necessary to support external lobby launching. See Tutorial 1: Connecting by Using the Lobby for a demonstation on how to connect an application by using a DirectPlay lobby.

Here are the basic steps necessary:

1. At startup, create an IDirectPlayLobby2 interface using the **CoCreateInstance** API.
2. The application can examine the connection information that was passed in by the lobby and modify some of the connection settings if necessary. The IDirectPlayLobby2::GetConnectionSettings method returns a DPLCONNECTION structure with the connection settings. This method returns DPERR_NOTLOBBIED if a lobby did not launch the application. New settings can be set with IDirectPlayLobby2::SetConnectionSettings.
3. IDirectPlayLobby2::Connect creates or joins the specified application session using the specified service provider, and returns an **IDirectPlay2** interface. Connect returns an error if the session could not be created/joined, or if a lobby didn't launch the application.
4. Use **QueryInterface** to obtain an IDirectPlay3 interface and then call **Release** on the **IDirectPlay2** interface.
5. Create a player using the name information supplied in the DPLCONNECTION structure obtained in step 2.

At this point, the application can continue on the same code path as if the user had manually selected a connection, joined or created a session, and entered the name of the player to create.

The IDirectPlayLobby2 interface can be saved if the game will pass information back to the lobby (see Lobby Messaging), or it can be discarded by using the **Release** method if no messages will be sent.

Many lobbies will launch the application and then go into a suspend mode waiting for the application to terminate. DirectPlay will notify the lobby when the application that it launched has terminated. For this reason it is very important that the application launched by the lobby not launch another application.

Lobby Messaging (Optional)

After registering a lobby-aware application and adding the code to support external lobby launching, the next step in integrating the application with the lobby is to send information back to the lobby or request information from the lobby. This is done by exchanging messages with the lobby through the IDirectPlayLobby2::SendLobbyMessage and IDirectPlayLobby2::ReceiveLobbyMessage methods on the IDirectPlayLobby2 interface. Standard message structures have been defined by DirectPlay to facilitate this functionality.

Sending information to the lobby is done through setting properties. The application must create and fill in a DPLMSG_SETPROPERTY structure and send it to the lobby by using the SendLobbyMessage method. Each property identifies a distinct type of data. The application developer should generate GUIDs (using Guidgen.exe) for every property that will be set. The lobby operator needs to obtain this list of GUIDs from the application developer along with the description of each property and the data structure of the property.

Properties can take the form of:

- Current individual player scores.
- Final individual player scores.
- Player configuration for later retrieval.
- Current game status (level or mission).

The lobby can store this information or display it to other users in the lobby so they can monitor the game's progress. An application can request confirmation that the property was set correctly by supplying a nonzero request ID with the set property message.

Requesting information from the lobby is done through requesting properties. The application must create and fill in a DPLMSG_GETPROPERTY structure and send it to the lobby by using the SendLobbyMessage method. At some later time, the lobby will send a message back to the application — a DPLMSG_GETPROPERTYRESPONSE structure that contains the property data that was requested. The application can retrieve the message from the lobby message queue using the ReceiveLobbyMessage method. As before, the application developer generates the GUIDs for each property and the lobby operator must obtain them from the developer.

The following list shows examples of some properties that can be requested from the lobby:

- Game configuration settings — enables players to configure the game in the lobby before launching the game.
- Player configuration settings — enables players to configure their players in the lobby or use a stored configuration from a previous session.
- Other information stored from a previous session.

Not all lobbies will be able to support these standard lobby messages. The application can determine if the lobby that it was launched from supports standard lobby messages by sending a DPLMSG_GETPROPERTY message requesting the DPLPROPERTY_MessagesSupported property.

DirectPlay Messages

The following sections describe how to use messages in DirectPlay.

- Synchronization
- Using System Messages
- Using Lobby Messages

Synchronization

An application can use two methods to process DirectPlay messages. The first is to check the receive queue during the main loop of the application. Typically, this means the application is single threaded.

Alternatively, an application can have a separate thread to wait for messages and process them. In this case the application should supply a non-NULL auto-reset synchronization event handle (see the Win32 **CreateEvent** API) when it creates players. DirectPlay will set this event whenever a message arrives for that player. All the local players in an application can share the same event or each can have his or her own event.

The message processing thread should then use the Win32 **WaitForSingleObject** API to wait until the event is set. Keep calling Receive until there are no more messages in the message queue.

Using System Messages

Messages returned by the IDirectPlay3::Receive method from player ID DPID_SYSMSG are system messages. All system messages begin with a double-word value specified by the **dwType** DWORD value. You can cast the buffer returned by the IDirectPlay3::Receive method to a generic message (DPMSG_GENERIC) and switch on the **dwType** element, which will have a value equal to one of the messages with the DPSYS_ prefix. After the application has determined which system message it is, the buffer should be cast to the appropriate structure (beginning with the DPMSG_ prefix) to read the data.

Your application should be prepared to handle the following system messages.

Value of dwType	Message structure	Cause
DPSYS_ADDGROUPTOGROUP	DPMSG_ADDGROUPTOGROUP	An existing group has been added to an existing group.
DPSYS_ADDPLAYERTOGROUP	DPMSG_ADDPLAYERTOGROUP	An existing player has been added to an existing group.
DPSYS_CHAT	DPMSG_CHAT	A chat message has been received.
DPSYS_CREATEPLAYERORGROUP	DPMSG_CREATEPLAYERORGROUP	A new player or group has been created.
DPSYS_DELETEGROUPFROMGROUP	DPMSG_DELETEGROUPFROMGROUP	A group has been removed from a group.
DPSYS_DELETEPLAYERFROMGROUP	DPMSG_DELETEPLAYERFROMGROUP	A player has been removed from a group.
DPSYS_DESTROYPLAYERORGROUP	DPMSG_DESTROYPLAYERORGROUP	An existing player or group has been destroyed.
DPSYS_HOST	DPMSG_HOST	The current host has left the session and this application is the new host.
DPSYS_SECUREMESSAGE	DPMSG_SECUREMESSAGE	A digitally signed or encrypted message has been received.
DPSYS_SESSIONLOST	DPMSG_SESSIONLOST	The connection with the session has been lost.
DPSYS_SETPLAYERORGROUPDATA	DPMSG_SETPLAYERORGROUPDATA	Player or group data has changed.
DPSYS_SETPLAYERORGROUPNAME	DPMSG_SETPLAYERORGROUPNAME	Player or group name has changed.
DPSYS_SETSESSIONDESC	DPMSG_SETSESSIONDESC	The session description has changed.
DPSYS_STARTSESSION	DPMSG_STARTSESSION	The lobby server is requesting that a session be started.

Using Lobby Messages

Messages returned by the IDirectPlayLobby2::ReceiveLobbyMessage method fall into two categories: DirectPlay-defined messages and custom-defined messages. The message category can be identified by the *lpdwMessageFlags* parameter of ReceiveLobbyMessage. The flags indicate that the message is either a system message (the DPLMSG_SYSTEM flag) or a standard message (the DPLMSG_STANDARD flag). If neither of these flags is set, the message is custom-defined. System messages are generated automatically by DirectPlay and sent only to the lobby to inform it of changes in the status of the application. Standard messages can be generated by either the lobby or the application and sent to the other.

The advantage of standard messages over custom-defined messages is that the receiver can positively interpret the message. It is not required that all applications or lobbies act on standard messages.

DirectPlay-defined messages all start with a DWORD value that identifies the type of the message. After retrieving a message using ReceiveLobbyMessage, the *lpData* pointer to the message data should be cast to the DPLMSG_GENERIC structure and the structure's **dwType** member examined. Based on the value of **dwType**, the *lpData* pointer should then be cast to the appropriate message structure for further processing.

Lobbies should be prepared to handle all the following message types. Applications need to handle the DPLMSG_GETPROPERTYRESPONSE message if they generate DPLMSG_GETPROPERTY messages.

Messages returned by the IDirectPlayLobby2::ReceiveLobbyMessage method that have a *dwFlags* parameter set to DPLMSG_SYSTEM are system messages. All system messages begin with a double-word value specified by **dwType**. You can cast the buffer returned by the IDirectPlayLobby2::ReceiveLobbyMessage method to a generic message (DPLMSG_GENERIC) and switch on the **dwType** element, which will have a value equal to one of the messages with the DPLSYS_ prefix.

The following list shows the possible values of the *dwType* data member, and the message structure and message cause associated with each value.

Value of dwType	Message structure	Cause
DPLSYS_APPTERMINATED	DPLMSG_GENERIC	The application has terminated.
DPLSYS_CONNECTIONSETTINGSREAD	DPLMSG_GENERIC	The application has read the connection settings.
DPLSYS_DPLAYCONNECTFAILED	DPLMSG_GENERIC	The application failed to connect to the DirectPlay session.
DPLSYS_DPLAYCONNECTSUCCEEDED	DPLMSG_GENERIC	The application successfully connected to the DirectPlay session.
DPLSYS_GETPROPERTY	DPLMSG_GETPROPERTY	The application is

		requesting a property from the lobby.
DPLSYS_GETPROPERTYRESPONSE	DPLMSG_GETPROPERTYRESPONSE	The lobby is responding to a prior DPLMSG_GETPROPERTY message.
DPLSYS_SETPROPERTY	DPLMSG_SETPROPERTY	The application is setting a property on the lobby.
DPLSYS_SETPROPERTYRESPONSE	DPLMSG_SETPROPERTYRESPONSE	The lobby is responding to a prior DPLMSG_SETPROPERTY message.

DirectPlay Address (Optional)

A DirectPlay Address is a data format that DirectPlay uses to pass addressing information between lobby servers, applications, DirectPlay, and service providers. The format is flexible enough to hold any number of variable-length data fields that make up a network address. It is not necessary to understand DirectPlay Addresses in order to use DirectPlay. In fact, it is possible to successfully write a DirectPlay application without understanding DirectPlay Addresses at all.

It is only necessary to learn about DirectPlay Addresses if you want to do the following things.

- Override the standard service provider dialog boxes. You need to create a DirectPlay Address that specifies which service provider to use and contains all the information the service provider needs to establish a connection. You then pass the DirectPlay Address to the IDirectPlay3::InitializeConnection method.
- Write a lobby client that will launch an external DirectPlay application. The lobby client needs to create a DirectPlay Address that specifies which service provider to use and contains all the information the service provider needs to establish a connection and automatically connect the client to a session. You need to create a DirectPlay Address of the session to be joined and put it in the DPLCONNECTION structure before calling the IDirectPlayLobby2::RunApplication method.

A DirectPlay Address is a sequence of variable-length data chunks tagged with a GUID that supply all the address information needed by DirectPlay. Examples are the network address of a server, the network address of a player, the email name of a player, or the network address of a session.

DirectPlay Address Data Types

Microsoft has predefined the following general data types for each data chunk. Based on the data type, the data must be cast to the appropriate type or structure to interpret the data.

GUID	Type of data.
DPAID_ComPort	A DPCOMPORTADDRESS structure that contains all the settings for the COM port. The serial connection service provider will use this information to configure the serial port.
DPAID_INet	A null-terminated ANSI string (LPSTR) containing an IP address ("137.55.100.173") or a domain net ("gameworld.com"). The length in bytes must include the terminator. A blank address is a string that contains only the ANSI terminator (0x00) and has a length of 1 byte. If a blank address is supplied, the Internet TCP/IP Connection service provider will use this information to enumerate sessions on the specified network address or broadcast on the subnet.
DPAID_INetW	A null-terminated Unicode string (LPWSTR) containing an IP address ("137.55.100.173") or a domain net ("gameworld.com"). The length in bytes must include the terminator. A blank address is a string that contains only the Unicode terminator (0x0000) and has a length of 2 bytes. If a blank address is supplied, the Internet TCP/IP Connection service provider will use this information to enumerate sessions on the specified network address or broadcast on the subnet.
DPAID_Modem	A variable-length null-terminated ANSI string (LPSTR) specifying which installed modem to use. The length in bytes must include the terminator. The modem service provider will use this modem without displaying a dialog box asking the user which modem to use. Use the IDirectPlay3::GetPlayerAddress method to determine which modems are available.
DPAID_ModemW	A variable-length null-terminated Unicode string (LPWSTR) specifying which installed modem to use. The length in bytes must include the terminator. The modem service provider will use this modem without displaying a dialog box asking the user which modem to use. Use the IDirectPlay3::GetPlayerAddress method to determine which modems are available.
DPAID_Phone	A variable-length null-terminated ANSI string (LPSTR) containing a phone number. The length in bytes must include the terminator. The modem service provider will call this phone number on the IDirectPlay3::EnumSessions method. If no modem is specified, the first modem will be used.
DPAID_PhoneW	A variable-length null-terminated Unicode string (LPWSTR) containing a phone number. The length in bytes must include the terminator. The modem service provider will call this phone number on the IDirectPlay3::EnumSessions method. If no modem is specified, the first modem will be used.
DPAID_ServiceProvider	The 16-byte GUID that specifies the service provider this DirectPlay Address applies to.

Using DirectPlay Addresses

You can use a DirectPlay Address to encapsulate all the information necessary to initialize a DirectPlay object. At a minimum, this is the GUID of a DirectPlay provider, but can also include the network address of an application or lobby server and even a specific session instance GUID.

A DirectPlay Address can be used to supply enough information to DirectPlay (and the DirectPlay provider) when it is initialized so that no dialog boxes appear later during the process of establishing a session or connecting to a session.

The DirectPlay Addresses returned by EnumConnections are simply registered DirectPlay providers that the user can choose from. When one of these is initialized, dialog boxes can appear asking the user for more information.

The application can create a DirectPlay Address directly using the IDirectPlayLobby2::CreateAddress and IDirectPlayLobby2::CreateCompoundAddress methods on the IDirectPlayLobby2 interface and pass the connection to the IDirectPlay3::InitializeConnection method to initialize the DirectPlay object. If enough valid information is supplied, then no DirectPlay dialog boxes will appear.

Examples of Using DirectPlay Addresses

This topic contains examples of DirectPlay Addresses and the data they contain for different connection types.

A DirectPlay Address describing an IPX connection

Initializing this connection will bind the DirectPlay object to the IPX service provider.

guidDataType	dwDataSize	Data
DPAID_ServiceProvider	16	{685BC400-9D2C-11cf-A9CD-00AA006886E3}

A DirectPlay Address describing a modem connection

Initializing this connection will bind DirectPlay to the modem service provider and store the phone number. A subsequent call the IDirectPlay3::EnumSessions will dial the number without asking the user for a phone number.

guidDataType	dwDataSize	Data
DPAID_ServiceProvider	16	{44EAA760-CB68-11cf-9C4E-00A0C905425E}
DPAID_Phone	9 (including NULL terminator)	"555-8237"

A DirectPlay Address describing a TCP/IP connection

Initializing this connection will bind DirectPlay to the TCP/IP service provider and store the IP address. A subsequent call IDirectPlay3::EnumSessions will enumerate sessions on this server without asking the user for an IP address.

guidDataType	dwDataSize	Data
DPAID_ServiceProvider	16	{36E95EE0-8577-11cf-960C-0080C7534E82}
DPAID_INet	10 (including NULL terminator)	"127.0.0.1"

Migrating from Previous Versions of DirectPlay

The DirectPlay version 5 API is fully compatible with applications written for any previous version of DirectPlay. That is, you can recompile your application by using DirectPlay on the DirectX 5 SDK without making any changes to the code. DirectPlay supplied with the DirectX 5 SDK supports all the APIs and behavior of earlier DirectPlay versions.

For specific information, see:

- Migrating from DirectPlay 3
- Migrating from DirectPlay 2 or Earlier

Migrating from DirectPlay 3

If you are migrating to DirectPlay version 5 from DirectPlay version 3, you do not have to make any changes. However, you should upgrade your application to use the new IDirectPlay3 or IDirectPlay3A interfaces.

For a list of new features in version 5 of DirectPlay, see What's New in DirectPlay 5.

Some system messages have changed for DirectPlay 5, with new data members added to the end. Applications that need to be backward compatible with older run times should either:

- compile with the DirectX 3 header files and libraries
- not reference new data members
- test with both the DirectX 3 and the DirectX 5 runtime

The system message structures that have new members for DirectPlay 5 are:

Structure	New Members
DPMSG_CREATEPLAYERORGROUP	DPID dpIdParent
	DWORD dwFlags
DPMSG_DESTROYPLAYERORGROUP	DPNAME dpnName
	DPID dpIdParent
	DWORD dwFlags

Migrating from DirectPlay 2 or Earlier

The names of the DirectPlay DLLs in version 3 and later are different from those in previous DirectPlay versions. Applications compiled with DirectX 2 or earlier do not use the new DirectPlay DLLs. To use the new DLLs, the application must be recompiled and linked to the Dplayx.lib import library.

It is also recommended that you add the code necessary to make the application lobby aware. This means that an external lobby or matchmaking program can start the application and supply it with all the information necessary to connect to a session. The application need not ask the user to decide on a service provider, select a session, or supply any other information (such as a telephone number or network address).

In addition, several other new features were added in the DirectPlay version 5 API, including the following:

- Internet support.
- Direct serial connection.
- More stability and robustness.
- Support for Unicode to better support localization.
- Host migration. If the host of a session drops out of the session, host responsibilities are passed on to another player. In DirectPlay version 2, if the host (name server) left a session, no new players could be created.
- Ability of the application to communicate with the lobby program to update games status for spectators, as well as receive information about initial conditions.
- Ability to host more than one application session on a computer.
- Ability to determine when a remote computer loses its connection and to generate appropriate messages.

There are other features in DirectPlay 5 that you can use to reduce the amount of communication-management code in your application, including the following:

- Ability to associate application-specific data with a DirectPlay group ID or player ID. This allows the application to leverage the player and group list-management code that is already part of DirectPlay. Local data is data that the local application uses directly, such as a bitmap that represents a player. Local data is not propagated over the network. Remote data is associated with the player or group. DirectPlay propagates any changes to remote data to all other applications in the session. Remote data must be shared among all the applications in a session, such as a player's position, orientation, and velocity. By using DirectPlay functions to propagate this data, the application need not manage it by using a series of methods that send and receive information.
- Ability for application to associate a name with a group. This is useful for team play.

Some of the new features added to DirectPlay 5 are not directly related to applications, including the following:

- APIs that the lobby client software uses to start and connect a lobby-able DirectPlay application. Also included are APIs that allow an application and the lobby to exchange information during a session.
- Service Provider development kit that includes documentation and sample code for creating your own service provider.

Migrating to the IDirectPlay3 Interface

To migrate an application created with DirectPlay version 2 or earlier, carry out the following steps:

1. Find out if your application was launched from a lobby client. For more information, see Step 2: Retrieving the Connection Settings.
2. If your application is enumerating service providers, use the EnumConnections method to determine if a service provider is available. If so, call the InitializeConnection method on the service provider. If InitializeConnection returns an error, the service provider cannot run on the system, and you should not add that service provider to the list to show to the user. If the call succeeds, use the Release method to release the DirectPlay object and add the service provider to the list.

3 Call the QueryInterface method on the **IDirectPlay** interface to obtain an IDirectPlay3 (Unicode) or IDirectPlay3A (ANSI) interface. The only difference between the two interfaces is how strings in the structures are read and written. For the Unicode interface, Unicode strings are read and written to the member of the structure that is of the **LPWSTR** type. For the ANSI interface, ANSI strings are read and written to the member of the structure that is of the **LPSTR** type.

4 Make all the changes necessary to use the new structures in existing APIs. For example, instead of the following:

```
lpDP->SetPlayerName(pidPlayer, lpszFriendlyName, lpszFormalName)
```

where *lpDP* is an **IDirectPlay** interface, use the following:

```
DPNAME PlayerName, *lpPlayerName;
PlayerName.dwSize = sizeof(DPNAME);
lpPlayerName = &PlayerName;

lpPayerName->lpszShortNameA = lpszFriendlyName;
lpPlayerName->lpszLongNameA = lpszFormalName;
lpDP3A->SetPlayerName(pidPlayer, lpPlayerName, 0)
```

where *lpDP3A* is an IDirectPlay3A interface. If the application is using Unicode strings (and therefore instantiates an IDirectPlay3 interface), use the following:

```
lpPayerName->lpszShortName = lpwszFriendlyName;
lpPlayerName->lpszLongName = lpwszFormalName;
lpDP3->SetPlayerName(pidPlayer, lpPlayerName, 0)
```

where *lpDP3* is an IDirectPlay3 interface.

5 Update the following system messages:

- DPSYS_ADDPLAYER has been replaced by DPSYS_CREATEPLAYERORGROUP.
- DPSYS_DELETEPLAYER and DPSYS_DELETEGROUP have been combined in a single DPSYS_DESTROYPLAYERORGROUP message.
- DPSYS_DELETEPLAYERFROMGRP has been changed to DPSYS_DELETEPLAYERFROMGROUP.

6 Update your application to generate a DPSYS_SETPLAYERORGROUPNAME message when a player or group name changes, and a DPSYS_SETPLAYERORGROUPDATA message when the player or group data changes.

7 Update the **DPSESSIONDESC** structure to DPSESSIONDESC2, and add the new members to the DPCAPS structure.

8 Update the callback functions for IDirectPlay3::EnumSessions, IDirectPlay3::EnumGroups, IDirectPlay3::EnumGroupPlayers, and IDirectPlay3::EnumPlayers.

9 Update the manner in which the *hEvent* parameter is supplied to the IDirectPlay3::CreatePlayer method. In previous versions of DirectPlay, this parameter was *lpEvent*. DirectPlay does not return an event; instead, the application must create it. This allows the application the flexibility of creating one event for all the players.

10 Set the DPSESSION_KEEPALIVE flag in the DPSESSIONDESC2 structure if the application needs DirectPlay to detect when players drop out of the game abnormally.

11 Update your application to create sessions with the DPSESSION_MIGRATEHOST flag. This enables another computer to become the host if the current host drops out of the session. If your application has any special code that only the host runs, then your application should set this flag when it creates a session. It should also add support for the DPSYS_HOST system message. For a list of system messages, see Using System Messages.

12 Become familiar with the new methods of the IDirectPlay3 interface and use them in your application. Pay particular attention to the IDirectPlay3::SetPlayerData and IDirectPlay3::GetPlayerData methods. You might be able to substitute the code where you broadcast player state information to all the other players by using the IDirectPlay3::Send and IDirectPlay3::Receive methods.

DirectPlay Tools and Samples

The following samples can be examined for examples of how to use DirectPlay. These DirectPlay samples are included on the DirectX 5 CD. The Control Panel tool can be used to find out about DirectPlay applications.

DirectX Control Panel Tool

- Found in Control Panel. Double-click to open the DirectX Properties dialog box.
- The DirectPlay tab displays all the registered DirectPlay service providers and applications.
- Open the DirectX Properties dialog box before starting a DirectPlay application to see on-the-fly statistics about DirectPlay communications such as bytes/second and messages/second.

SDK/SAMPLES/BELLHOP

- A lobby client program the uses the IDirectPlay3 interface to communicate with a lobby server.
- Uses the LSERVER test lobby server so that a real lobby client/server environment can be set up.
- BELLHOP can be used to test external lobby support in your application.
- LSERVER can be used as a test server to write your own lobby client or integrate a lobby client into your application.

SDK/SAMPLES/DPCHAT

- A simple Windows-based chat program that uses IDirectPlay3
- Uses IDirectPlayLobby to make it lobby-able

SDK/SAMPLES/DPLAUNCH

- A stand-alone application that demonstrates how a DirectPlay 5 application can be launched from an external source using the IDirectPlayLobby2 interface
- Can also be used to test the lobby support in your application

SDK/SAMPLES/DPSLOTS

- A DirectPlay client/server application that uses security.
- The server controls all the slot machines and tracks how much money each player has.
- The client securely logs in to the server.

SDK/SAMPLES/DUEL

- Multiplayer game that uses the IDirectPlay interface
- Uses IDirectPlayLobby to make it lobby-able

SDK/SAMPLES/DXVIEW

- An application that shows all the available service providers and their capabilities.
- Shows all the registered lobby-aware applications installed on the system.
- Uses asynchronous EnumSessions to monitor the active sessions on the service provider.

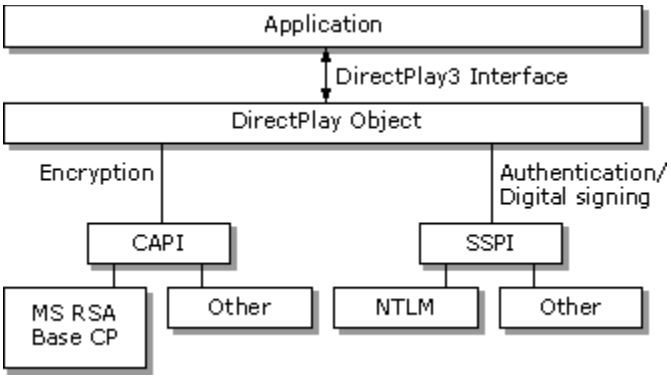
SDK/SAMPLES/OVERRIDE

- A simple demonstration of how to override the DirectPlay service provider dialogs.

Security and Authentication

DirectPlay security allows an application running on a server to authenticate users against a database of known users before allowing them to join a session. Once a user has been authenticated, all communications between the client and server can be done securely either by digitally signing messages (to verify the identity of the sender) or by encrypting the messages.

The following diagram shows DirectPlay security architecture. (SSPI = Security Support Provider Interface, CAPI = CryptoAPI, MS RSA Base CP = Microsoft RSA Base Cryptographic Provider, NTLM = NT LAN Manager.)



User and Message Authentication

DirectPlay provides user and message authentication (digital signing) support through the Windows Security Support Provider Interface (SSPI). This is a standard interface that gives software access to various security packages under the Windows Operating System. A security package is an implementation of a protocol between a client and server that establishes the identity of the client and provides other security services, such as digital signing. The default security package that DirectPlay uses is called NTLM (Windows NT LAN Manager) Security Support Provider.

This security package is based on the NTLM authentication protocol. NTLM is a shared-secret, user challenge-response authentication protocol that supports pass-through authentication to a domain controller in the server's domain or in a domain trusted by the current domain's domain controller. This protocol provides a high level of security, as passwords are never sent out on the network. NTLMSSP ships with the Windows 95 and Windows NT operating systems.

A DirectPlay application can choose to use a different SSPI security package when it creates a session by calling the SecureOpen method. For example, DPA (Distributed Password Authentication) Security Support Provider is another security package that organizations can use to provide membership services to a large customer base (hundreds of thousands). This security package is available through the Microsoft Commercial Internet Services (MCIS) Membership Server.

Message Privacy (encryption/decryption)

DirectPlay provides message encryption support through the Windows Cryptography Application Programming Interface (CAPI). This is a standard interface similar to SSPI that gives software access to various cryptographic packages under the Windows Operating System. This architecture allows DirectPlay applications to plug in cryptographic packages that provide the desired level of encryption (40 bit, 128 bit, and so on) legally allowed in the locale of use.

The default CryptoAPI (CAPI) provider for DirectPlay cryptography services is the Microsoft RSA Base Cryptographic Provider v. 1.0. The default CAPI provider type is PROV_RSA_FULL. The default encryption algorithm is the CALG_RC4 stream cipher. This provider is supplied by Microsoft and is included with Internet Explorer for Windows 95, Windows 95 OSR-2 Update, and Windows NT 4.0 operating system.

A DirectPlay application can choose to use a different Cryptographic provider when it creates a session by using the SecureOpen method. Please note that DirectPlay only supports stream ciphers.

For more information about SSPI, NTLM, DPA, MCIS, CAPI, and the RSA Base Cryptographic Provider, see http://www.microsoft.com.

Secure Sessions

User authentication should not be confused with password protection of a session. Authentication is used to verify that the user is allowed access to the server by virtue of having been added to the membership database by the administrator of the server. Only users that are part of the membership database are permitted to join the session. A password can be added to a session, so that only those members who know the password can join a particular session. Additionally, authentication requires a server that supports authentication, while any computer can put a password on a session.

For example, a server on the Internet might have a membership of a thousand users. Anybody can enumerate the sessions that are available on the server but only members will be able to join sessions on the server. Users who want only their friends (who are members) to be able to join can put a password on their session.

Once a secure server has been set up and an initial membership list has been established, a secure DirectPlay session can be started on it. Creating a secure DirectPlay session simply requires the server to create a session using IDirectPlay3::Open or IDirectPlay3::SecureOpen and specify the DPSESSION_SECURESERVER flag in the DPSESSIONDESC2 structure.

A DirectPlay application can choose to use alternate providers when it creates a session by calling the SecureOpen method and specifying the providers to use in the DPSECURITYDESC structure. For a different SSPI provider (for user and message authentication), an application needs to specify the provider name in the *lpszSSPIProvider* member of the DPSECURITYDESC structure. For a different CryptoAPI provider (for message privacy), an application needs to specify the provider name, provider type, and encryption algorithm in the *lpszCAPIProvider*, *dwCAPIProviderType*, and *dwEncryptionAlgorithm* members respectively.

When a client enumerates this session, the DPSESSIONDESC2 structure returned by IDirectPlay3::EnumSessions will have the DPSESSION_SECURESERVER flag set. This tells the client that authentication credentials will be required to join the session. If the client attempts to join the session using IDirectPlay3::Open, the security package may allow the user in if the user's credentials were already established through a system logon (for example, through NT LAN Manager). Otherwise, a DPERR_LOGONDENIED error is returned. The application must then collect credentials from the user, put them in a DPCREDENTIALS structure, and try to join the session again by calling SecureOpen, passing in the DPCREDENTIALS structure. SecureOpen is recommended for security.

Within a secure session, all DirectPlay system messages are digitally signed to verify the identity of the sender. Certain system messages that carry sensitive information are encrypted. System messages that originate from one player and need to be broadcast to all the other players in the session are first sent to the server. The server then puts its signature on the message and forwards the message to all the other computers in the session. Player to player messages are not signed by default and are not routed through the server.

An application can choose to sign or encrypt specific player-to-player messages using the DPSEND_SIGNED and DPSEND_ENCRYPTED flags in the IDirectPlay3::Send method. Signed and encrypted player messages are routed through the server and delivered as secure system messages. When a secure message is received by a player, the message will contain flags indicating whether the message came in signed or encrypted. Messages that don't pass the verification are dropped.

DirectPlay Interface Overviews

DirectPlay is composed of objects and interfaces based on the component object model (COM). COM is a foundation for an object-based system that focuses on the reuse of interfaces, and it is the model at the heart of OLE programming. It is also an interface specification from which any number of interfaces can be built.

New methods and functionality are exposed in COM through the use of new interfaces that are sequentially numbered. The current DirectPlay interface is called IDirectPlay3. The old interfaces, **IDirectPlay** and **IDirectPlay2**, still exist for backward compatibility with applications written to those interfaces.

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

- Unicode vs. ANSI Interfaces
- IDirectPlay Interface
- IDirectPlay2 Interface
- IDirectPlay3 Interface
- IDirectPlayLobby Interface
- IDirectPlayLobby2 Interface

Unicode vs. ANSI Interfaces

DirectPlay supports both Unicode and ANSI strings by defining string pointers in a structure as the union of a Unicode string pointer (**LPWSTR**) and an ANSI string pointer (**LPSTR**). The two string pointers have different names. Typically, the ANSI member ends with the letter "A". Depending on which **IDirectPlay** interface is chosen (IDirectPlay3 for Unicode or IDirectPlay3A for ANSI), or which IDirectPlayLobby2 interface is chosen (IDirectPlayLobby2 for Unicode or **IDirectPlayLobby2A** for ANSI), the application should read and write the appropriate strings from the structure and ignore the other one.

IDirectPlay Interface

The **IDirectPlay** COM interface remains part of DirectPlay version 5. It contains the methods required to run applications that were written for the DirectX SDK versions 1 and 2. Although you could use this interface to create new applications, it is recommended that you use the newer DirectPlay interfaces, IDirectPlay3 and IDirectPlay3A, to take advantage of their increased functionality.

IDirectPlay2 Interface

The **IDirectPlay2** COM interface remains part of DirectPlay version 5. It contains the methods required to run applications that were written for the DirectX SDK versions 3. Although you could use this interface to create new applications, it is recommended that you use the newer DirectPlay interfaces, IDirectPlay3 and IDirectPlay3A, to take advantage of their increased functionality.

IDirectPlay3 Interface

This is the latest interface. IDirectPlay3 directly inherits from **IDirectPlay2** so any code written for **IDirectPlay2** will work with IDirectPlay3 with no modification.

The new methods in IDirectPlay3 are:

- AddGroupToGroup
- CreateGroupInGroup
- DeleteGroupFromGroup
- EnumConnections
- EnumGroupsInGroup
- GetGroupConnectionSettings
- GetGroupFlags
- GetGroupParent
- GetPlayerAccount
- GetPlayerFlags
- InitializeConnection
- SecureOpen
- SendChatMessage
- SetGroupConnectionSettings
- StartSession

IDirectPlayLobby Interface

The **IDirectPlayLobby** COM interface remains part of DirectPlay version 5. It contains the methods required to run applications that were written for the DirectX SDK versions 1 and 2. Although you could use this interface to create new applications, it is recommended that you use the newer DirectPlay interface, IDirectPlayLobby2, to take advantage of its increased functionality.

IDirectPlayLobby2 Interface

The IDirectPlayLobby2 interface lets game developers launch external applications, enable communication between an application and a lobby client, and manipulate DirectPlay Addresses.

IDirectPlayLobby2 supports all the methods of **IDirectPlayLobby**. See Building Lobby-Aware Applications for information about detecting a launch by an external lobby and registering lobby-aware applications. See DirectPlay Lobby Overview for general information about lobby sessions.

The new method in IDirectPlayLobby2 is:

- CreateCompoundAddress

DirectPlay Tutorials

This section contains four tutorials that provide step-by-step instructions about how to connect an application with or without a lobby, how to override service provider dialog boxes, and how to create a self-refreshing session list.

The first tutorial demonstrates how to connect an application by using a DirectPlay lobby. The second tutorial demonstrates how to connect an application by using a dialog box that queries the user for connection information. You should write your application so that it can start by using either method. The code is available in the DPCHAT sample in the LOBBY.CPP and DIALOG.CPP files.

The third tutorial demonstrates the calls you need to supply the service provider with all the information it needs so that it doesn't display dialog boxes to the user requesting information.

The fourth tutorial demonstrates how to create a self-refreshing session list.

- Tutorial 1: Connecting by Using the Lobby (DPCHAT)
- Tutorial 2: Connecting by Using a Dialog Box (DPCHAT)
- Tutorial 3: Overriding the Service Provider Dialogs
- Tutorial 4: Creating Self-Refreshing Session Lists

Note The sample files in these tutorials are written in C++. If you are using a C compiler, you must make the appropriate changes to the files for them to successfully compile. At the very least, you must add the vtables and **this** pointers to the interface methods. For more information, see .

Tutorial 1: Connecting by Using the Lobby

An application written to use the IDirectPlayLobby2 interface can be connected to a session without requiring the user to manually enter connection information in a dialog box. To demonstrate how to create a lobbied application, the DPCHAT sample performs the following steps:

- Step 1: Creating a DirectPlayLobby Object
- Step 2: Retrieving the Connection Settings
- Step 3: Configuring the Session Description
- Step 4: Connecting to a Session
- Step 5: Creating a Player

Step 1: Creating a DirectPlayLobby Object

To use a DirectPlay lobby, you first create an instance of a DirectPlayLobby object by calling the DirectPlayLobbyCreate function. This function contains five parameters. The first, third, and fourth parameters are always set to NULL and are included for future expansion. The second parameter contains the address of a pointer that identifies the location of the DirectPlayLobby object if it is created. The fifth parameter is always set to 0, and is also included for future expansion.

The following example shows one way to create a DirectPlayLobby object:

```
// Get an ANSI DirectPlay lobby interface.
hr = DirectPlayLobbyCreate(NULL, &lpDirectPlayLobbyA, NULL, NULL, 0);
if FAILED(hr)
    goto FAILURE;
```

Step 2: Retrieving the Connection Settings

After the DirectPlayLobby object has been created, use the IDirectPlayLobby2::GetConnectionSettings method to retrieve the connection settings returned from the lobby. If this method returns DPERR_NOTLOBBIED, the lobby did not start this application and the user will have to configure the connection manually. If any other error occurs, your application should report an error that indicates that lobbying the application failed.

The following example shows how to retrieve the connection settings:

```
// Retrieve the connection settings from the lobby.
// If this routine returns DPERR_NOTLOBBIED, then a lobby did not
// start this application and the user needs to configure the
// connection.

// Pass a NULL pointer to retrieve only the size of the
// connection settings
hr = lpDirectPlayLobbyA->GetConnectionSettings(0, NULL, &dwSize);
if (DPERR_BUFFERTOOSMALL != hr)
    goto FAILURE;

// Allocate memory for the connection settings.
lpConnectionSettings = (LPDPLCONNECTION) GlobalAllocPtr(GHND, dwSize);
    if (NULL == lpConnectionSettings)
    {
        hr = DPERR_OUTOFMEMORY;
        goto FAILURE;
    }

    // Retrieve the connection settings.
    hr = lpDirectPlayLobbyA->GetConnectionSettings(0,
        lpConnectionSettings, &dwSize);
    if FAILED(hr)
        goto FAILURE;
```

Step 3: Configuring the Session Description

You should examine the DPSESSIONDESC2 structure to ensure that all the flags and properties that your application needs are set properly. If modifications are necessary, store the modified connection settings by using the IDirectPlayLobby2::SetConnectionSettings method.

The following example shows how to configure the session description and set the connection settings:

```
// Before the game connects, it should configure the session
// description with any settings it needs.

// Set the flags and maximum players used by the game.
lpConnectionSettings->lpSessionDesc->dwFlags = DPSESSION_MIGRATEHOST |
    DPSESSION_KEEPALIVE;
lpConnectionSettings->lpSessionDesc->dwMaxPlayers = MAXPLAYERS;

// Store the updated connection settings.
hr = lpDirectPlayLobbyA->SetConnectionSettings(0, 0,
        lpConnectionSettings);
if FAILED(hr)
    goto FAILURE;
```

Step 4: Connecting to a Session

After the session description is properly configured, your application can use the IDirectPlayLobby2::Connect method to start and connect itself to a session. If this method returns DP_OK, you can create one or more players. If it returns DPERR_NOTLOBBIED, the user will have to manually select a communication medium for your application. (You can identify the service providers installed on the system by using the DirectPlayEnumerate function.) If any other error value is returned, your application should report an error that indicates that lobbying the application failed.

The following example shows how to connect to a session:

```
// Connect to the session. Returns an ANSI IDirectPlay3A interface.
hr = lpDirectPlayLobbyA->Connect(0, &lpDirectPlay2A, NULL);
if FAILED(hr)
    goto FAILURE;
// Obtain an IDriectPlay3A interface
hr= lpDirectPlay2A->QueryInterface(IID_DirectPlay3A,
(LPVOID*)&lpDirectPlay3A);
if FAILED(hr)
    goto FAILURE;
```

Step 5: Creating a Player

If the application was successfully started by using the IDirectPlayLobby2::Connect method, it can now create one or more players. It can use the IDirectPlay3::CreatePlayer method to create a player with the name specified in the DPNAME structure (which was filled in by the IDirectPlayLobby2::GetConnectionSettings method).

The following example shows how to create a player:

```
// create a player with the name returned in the connection settings
hr = lpDirectPlay3A->CreatePlayer(&dpidPlayer,
    lpConnectionSettings->lpPlayerName,
    lpDPInfo->hPlayerEvent, NULL, 0, 0);
if FAILED(hr)
    goto FAILURE;
```

Now your application is connected and you are ready to play.

Tutorial 2: Connecting by Using a Dialog Box

If a lobby did not start your application, you should include code that allows the user to manually enter the connection information. To demonstrate how to manually connect to the session and create one or more players, the DPCHAT sample performs the following steps:

- Step 1: Creating the DirectPlay Object
- Step 2: Enumerating and Initializing the Service Providers
- Step 3: Joining a Session
- Step 4: Creating a Session
- Step 5: Creating a Player

Step 1: Creating the DirectPlay Object

Before any methods can be called, the application must create an interface to a DirectPlay object.

The following example shows how the create the IDirectPlay3A interface:

```
HRESULT CreateDirectPlayInterface( LPDIRECTPLAY3A *lplpDirectPlay3A )
{
   HRESULT         hr;
   LPDIRECTPLAY3A  lpDirectPlay3A = NULL;

   // Create an IDirectPlay3 interface
   hr = CoCreateInstance( CLSID_DirectPlay, NULL, CLSCTX_INPROC_SERVER,
                          IID_IDirectPlay3A, (LPVOID*)&lpDirectPlay3A);

   // Return interface created
   *lplpDirectPlay3A = lpDirectPlay3A;

   return (hr);
}
```

Step 2: Enumerating and Initializing the Service Providers

The next step in creating a manual connection is to request that the user select a communication medium for the application. Your application can identify the service providers installed on a personal computer by using the EnumConnections method.

The following example shows how to enumerate the service providers:

```
lpDirectPlay3A->EnumConnections(&DPCHAT_GUID,
DirectPlayEnumConnectionsCallback, hWnd, 0);
```

The second parameter in the EnumConnections method is a callback that enumerates service providers registered with DirectPlay. The following example shows one possible way of implementing this callback function:

```
BOOL FAR PASCAL DirectPlayEnumConnectionsCallback(
     LPCGUID lpguidSP, LPVOID lpConnection, DWORD dwConnectionSize,
     LPCDPNAME lpName, DWORD dwFlags, LPVOID lpContext)
{
   HWND     hWnd = (HWND) lpContext;
   LRESULTiIndex;
   LPVOID   lpConnectionBuffer;

   // Store service provider name in combo box
   iIndex = SendDlgItemMessage(hWnd, IDC_SPCOMBO, CB_ADDSTRING, 0,
       (LPARAM) lpName->lpszShortNameA);
   if (iIndex == CB_ERR)
     goto FAILURE;

   // make space for connection
   lpConnectionBuffer = GlobalAllocPtr(GHND, dwConnectionSize);
   if (lpConnectionBuffer == NULL)
     goto FAILURE;

   // Store pointer to connection  in combo box
   memcpy(lpConnectionBuffer, lpConnection, dwConnectionSize);
   SendDlgItemMessage(hWnd, IDC_SPCOMBO, CB_SETITEMDATA, (WPARAM) iIndex,
       (LPARAM) lpConnectionBuffer);

FAILURE:
    return (TRUE);
}
```

Once the user selects which connection to use, the DirectPlay object must be initialized with the connection buffer associated with it.

```
hr = lpDirectPlay3A->InitializeConnection(lpConnection, 0);
```

Step 3: Joining a Session

If the user wants to join an existing session, enumerate the available sessions by using the IDirectPlay3::EnumSessions method, present the choices to the user, and then connect to that session by using the IDirectPlay3::Open method, specifying the DPOPEN_JOIN flag. The service provider might display a dialog box requesting some information from the user before it can enumerate the sessions.

See Tutorial 4 for details on the asynchronous EnumSessions functionality.

The following example shows how to enumerate the available sessions:

```
// Search for this kind of session.
ZeroMemory(&sessionDesc, sizeof(DPSESSIONDESC2));
sessionDesc.dwSize = sizeof(DPSESSIONDESC2);
sessionDesc.guidApplication = DPCHAT_GUID;

hr = lpDirectPlay3A->EnumSessions(&sessionDesc, 0, EnumSessionsCallback,
    hWnd, DPENUMSESSIONS_AVAILABLE);
if FAILED(hr)
    goto FAILURE;
```

In the previous example, the third parameter in the IDirectPlay3A::EnumSessions method is a callback that enumerates the available sessions. The following example shows one way to implement this callback function:

```
BOOL FAR PASCAL EnumSessionsCallback(
    LPCDPSESSIONDESC2 lpSessionDesc, LPDWORD lpdwTimeOut,
    DWORD dwFlags, LPVOID lpContext)
{
HWND   hWnd = lpContext;
LPGUID lpGuid;
LONG   iIndex;

// Determine if the enumeration has timed out.
if (dwFlags & DPESC_TIMEDOUT)
    return (FALSE);            // Do not try again

// Store the session name in the list.
iIndex = SendDlgItemMessage(hWnd, IDC_SESSIONLIST, LB_ADDSTRING,
    (WPARAM) 0, (LPARAM) lpSessionDesc->lpszSessionNameA);
if (iIndex == CB_ERR)
    goto FAILURE;

// Make space for the session instance GUID.
lpGuid = (LPGUID) GlobalAllocPtr(GHND, sizeof(GUID));
if (lpGuid == NULL)
    goto FAILURE;

// Store the pointer to the GUID in the list.
*lpGuid = lpSessionDesc->guidInstance;
SendDlgItemMessage(hWnd, IDC_SESSIONLIST, LB_SETITEMDATA,
    (WPARAM) iIndex, (LPARAM) lpGuid);

FAILURE:
    return (TRUE);
}
```

After the user has selected a session, your application can allow the user to join an existing session. The following example shows how to join an existing session:

```
// Join an existing session.
```

```
ZeroMemory(&sessionDesc, sizeof(DPSESSIONDESC2));
sessionDesc.dwSize = sizeof(DPSESSIONDESC2);
sessionDesc.guidInstance = *lpguidSessionInstance;

hr = lpDirectPlay3A->Open(&sessionDesc, DPOPEN_JOIN);
if FAILED(hr)
    goto OPEN_FAILURE;
```

Step 4: Creating a Session

If the user wants to create a new session, your application can create it by using the IDirectPlay3::Open method and specifying the DPOPEN_CREATE flag. Again, the service provider might display a dialog box requesting information from the user before it can create the session.

The following example shows how to create a new session:

```
// Host a new session.
ZeroMemory(&sessionDesc, sizeof(DPSESSIONDESC2));
sessionDesc.dwSize = sizeof(DPSESSIONDESC2);
sessionDesc.dwFlags = DPSESSION_MIGRATEHOST | DPSESSION_KEEPALIVE;
sessionDesc.guidApplication = DPCHAT_GUID;
sessionDesc.dwMaxPlayers = MAXPLAYERS;
sessionDesc.lpszSessionNameA = lpszSessionName;

hr = lpDirectPlay3A->Open(&sessionDesc, DPOPEN_CREATE);
if FAILED(hr)
    goto OPEN_FAILURE;
```

Step 5: Creating a Player

After a session has been created or joined, your application can create one or more players by using the IDirectPlay3::CreatePlayer method. The following example shows one way to create a player:

```
// Fill out the name structure.
ZeroMemory(&dpName, sizeof(DPNAME));
dpName.dwSize = sizeof(DPNAME);
dpName.lpszShortNameA = lpszPlayerName;
dpName.lpszLongNameA = NULL;

// Create a player with this name.
hr = lpDirectPlay3A->CreatePlayer(&dpidPlayer, &dpName,
    lpDPInfo->hPlayerEvent, NULL, 0, 0);
if FAILED(hr)
    goto CREATEPLAYER_FAILURE;
```

Your application can determine a player's communication capabilities by using the IDirectPlay3::GetCaps and IDirectPlay3::GetPlayerCaps methods. Your application can find other players by using the IDirectPlay3::EnumPlayers method.

Now your application is connected and you are ready to play.

Tutorial 3: Overriding the Service Provider Dialogs

DirectPlay now gives applications the ability to suppress the standard service provider dialogs. Below is a brief outline of how this is to be done. A code example can be found in the OVERRIDE sample application.

It is generally not possible to suppress all service provider dialogs. The standard TCP/IP, modem, and serial service provider dialogs can be suppressed (IPX has no dialog box). However, there is the possibility that third party service providers might require fairly complex information from the user which cannot be overridden in any general way. The solution is to simply allow these dialog boxes to appear over your application user interface. If the application is a DirectDraw full-screen application, be sure to turn off page flipping before calling IDirectPlay3::EnumSessions or IDirectPlay3::Open to create a session.

Another way to suppress service provider dialog boxes is to make your application lobby-aware. Most third party service providers will also have a lobby from which to launch games, and games launched from a lobby do not display a connection dialog box.

An application first calls IDirectPlay3::EnumConnections to see what connections are available, presents the list to the user, and allows the user to select one. Once the user has selected one, the application can attempt to override the dialog box before calling IDirectPlay3::EnumSessions or IDirectPlay3::Open.

These are the steps you should follow to suppress service provider dialog boxes:

1. Examine the service provider GUID of the selected service provider to see if it matches one of the known service providers that the application knows how to override. If the service provider GUID is unknown then skip the remaining steps and be prepared to allow a dialog box to appear.
2. Display the appropriate user interface to collect the information needed from the user for that specific service provider. **IPX** requires no information. **TCP/IP** requires an IP address for **EnumSessions** (DPAID_Inet or DPAID_InetW) but requires nothing to create a session using **Open**. **Modem-to-modem** requires the user to select a modem (DPAID_Modem or DPAID_ModemW), and also needs a phone number (DPAID_Phone or DPAID_PhoneW) when calling **EnumSessions**. **Serial link** needs the DPCOMPORTADDRESS structure (DPAID_ComPort) filled in to configure the COM port for both **EnumSessions** and **Open**.
3. Build a DirectPlay Address using the IDirectPlayLobby2::CreateCompoundAddress method. The address elements that must be passed in are the service provider GUID (DPAID_ServiceProvider) and the individual address components for the selected service provider.
4. Initialize the DirectPlay object by calling InitializeConnection with the DirectPlay Address.
5. Call **EnumSessions** with the DPENUMSESSIONS_RETURNSTATUS flag. This will prevent any status dialog boxes from appearing and, if the connection cannot be made immediately, **EnumSessions** will return with a DPERR_CONNECTING error. Your application must periodically call **EnumSessions** until DP_OK is returned (meaning the enumeration was successful) or some other error is returned (meaning it failed).
6. If a session is to be created using the **Open** method with DPOPEN_CREATE, specify the DPOPEN_RETURNSTATUS flag as well. Like DPENUMSESSIONS_RETURNSTATUS, this will suppress status dialog boxes and return DPERR_CONNECTING until the function is complete.

Note: In some cases, the application will need to query the service provider at runtime to obtain a list of valid choices for a particular DirectPlay Address element. For example, to obtain a list of the modems installed in the system. The application must create a separate DirectPlay object, initialize the modem service provider and then call IDirectPlay3::GetPlayerAddress with a DPID of zero to obtain a DirectPlay Address that will contain the list of modems. After releasing the DirectPlay object, the application must parse the address using IDirectPlayLobby2::EnumAddress and extract the modem list to present to the user.

Tutorial 4: Creating Self-Refreshing Session Lists

IDirectPlay3::EnumSessions can now be called asynchronously. This gives an application the ability to maintain a self-refreshing session list. A code example can be found in the DUEL and DXVIEW sample applications.

The steps you need to follow to create a self-refreshing session list are:

1. Call IDirectPlay3::EnumSessions with the DPENUMSESSIONS_ASYNC flag and a time-out of zero (which will use the service provider default). The method will not enumerate any sessions and will return immediately. However, DirectPlay is enumerating sessions in the background.
2. Display the user interface in which all the sessions will appear. Set a timer to go off at whatever interval you want to refresh your session list. The application can find out what the default time-out interval of the enumeration is by calling IDirectPlay3::GetCaps.
3. Each time the timer goes off, call EnumSessions to obtain the current session list. This is a complete active session list with stale sessions deleted, new sessions added, and existing sessions updated. Delete all the items from the list before calling EnumSessions and add the sessions back to the list in the EnumSessionsCallback2 function.

DirectPlayCreate

This function is obsolete and remains for compatibility with applications written using DirectX 3. It is recommended that applications create the desired DirectPlay interface directly using **CoCreateInstance**. By using **CoCreateInstance** you can obtain an IDirectPlay3 interface directly rather than getting an **IDirectPlay** interface, having to use **QueryInterface** to access an IDirectPlay3 interface, and releasing the **IDirectPlay** interface.

Creates a new DirectPlay object and obtains an **IDirectPlay** interface pointer. If the application supplies GUID_NULL for the *lpGUIDSP* parameter, this function creates a DirectPlay object but does not initialize a service provider. The application can then call the IDirectPlay3::InitializeConnection method to initialize the service provider or lobby provider.

The application can supply a service provider GUID (see DirectPlayEnumerate) to indicate which service provider to bind.

In order to use the latest DirectPlay functionality, the application must obtain an IDirectPlay3 or IDirectPlay3A interface pointer using the **QueryInterface** method.

HRESULT WINAPI DirectPlayCreate(
LPGUID *lpGUIDSP*,
LPDIRECTPLAY FAR **lplpDP*,
IUnknown **lpUnk*
);

Parameters

lpGUIDSP
Pointer to the GUID of the service provider that the DirectPlay object should be initialized with. Pass in a pointer to GUID_NULL to create an uninitialized DirectPlay object.

lplpDP
Pointer to an interface pointer to be initialized with a valid IDirectPlay interface. The application will need to use the QueryInterface method to obtain an **IDirectPlay2**, IDirectPlay3 (UNICODE strings) or **IDirectPlay2A**, IDirectPlay3A (ANSI strings) interface.

lpUnk
Pointer to the containing *IUnknown*. This parameter is provided for future compatibility with COM aggregation features. Presently, however, the **DirectPlayCreate** function returns an error if this parameter is anything but NULL.

Return Values

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

CLASS_E_NOAGGREGATION
DPERR_EXCEPTION
DPERR_INVALIDPARAMS
DPERR_UNAVAILABLE

DPERR_UNAVAILABLE is returned if a DirectPlay object could not be created.
DPERR_INVALIDPARAMS is returned if the GUID provided is invalid.

Remarks

This function attempts to initialize a DirectPlay object and sets a pointer to it if successful. Your application should call the DirectPlayEnumerate function immediately before initialization to determine what types of service providers are available (the DirectPlayEnumerate function fills in the *lpGUIDSP* parameter of **DirectPlayCreate**).

This function returns a pointer to an **IDirectPlay** interface. The current interfaces for DirectX 5 are IDirectPlay3 and IDirectPlay3A, which need to be obtained through a call to the **QueryInterface** method on the **IDirectPlay** interface returned by **DirectPlayCreate**.

See Also

IDirectPlay3::InitializeConnection, DirectPlayEnumerate

DirectPlayEnumerate

Enumerates the DirectPlay service providers installed on the system.

This function is obsolete and remains for compatibility with applications written using DirectX 3. It will only enumerate service providers, not DirectPlay Addresses (connections). It is recommended that applications use the IDirectPlay3::EnumConnections method to enumerate all the connections available to the application after creating an IDirectPlay3 interface.

This function will not enumerate service providers that have the **Enumerate** value in the registry set to zero. For backward compatibility, this function will only return simple service providers registered under the "Service Providers" registry key.

HRESULT WINAPI DirectPlayEnumerate(
LPDPENUMDPCALLBACK *lpEnumCallback*,
LPVOID *lpContext*
);

Parameters

lpEnumCallback

Pointer to a callback function that will be called with a description of each DirectPlay service provider installed in the system. Depending on whether UNICODE is defined or not, the prototype for the callback function will have the service provider name *lpSPName* defined as a LPWSTR (for Unicode) or LPSTR (for ANSI).

lpContext

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

Return Values

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

DPERR_EXCEPTION
DPERR_GENERIC
DPERR_INVALIDPARAMS

DPERR_INVALIDPARAMS is returned if an invalid enumeration callback was supplied.

Remarks

This function will enumerate service providers installed in the system even though the system might not be capable of using those service providers. For example, a TAPI service provider will be part of the enumeration even though the system might not have a modem installed.

See Also

IDirectPlay3::EnumConnections, DirectPlayCreate

DirectPlayLobbyCreate

Creates an instance of a DirectPlayLobby object. This function attempts to initialize a DirectPlayLobby object and set a pointer to it.

HRESULT WINAPI DirectPlayLobbyCreate(
LPGUID *lpGUIDSP*,
LPDIRECTPLAYLOBBY **lplpDPL*,
IUnknown **lpUnk*,
LPVOID *lpData*,
DWORD *dwDataSize*
);

Parameters

lpGUIDSP
Reserved for future use; must be set to NULL.

lplpDPL
Pointer to a pointer to be initialized with a valid **IDirectPlayLobby** interface. To get an IDirectPlayLobby2 interface, use this function to get an **IDirectPlayLobby** interface, then call:

```
IDirectPlayLobby->QueryInterface( IID_IDirectPlayLobby2, (LPVOID*) &lpDP2 );
```

lpUnk
Pointer to the containing *IUnknown* interface. This parameter is provided for future compatibility with COM aggregation features. Presently, however, **DirectPlayLobbyCreate** returns an error if this parameter is anything but NULL.

lpData
Extra data needed to create the DirectPlayLobby object. This parameter must be set to NULL.

dwDataSize
This parameter must be set to zero.

Return Values

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

CLASS_E_NOAGGREGATION
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY

EnumAddressCallback

Application-defined callback function for the IDirectPlayLobby2::EnumAddress method.

BOOL FAR PASCAL EnumAddressCallback(
REFGUID *guidDataType*,
DWORD *dwDataSize*,
LPCVOID *lpData*,
LPVOID *lpContext*
);

Parameters

guidDataType
Pointer to a globally unique identifier (GUID) indicating the type of this data chunk. For example, this parameter might be &DPAID_Phone or &DPAID_INet. (In C++, it is a reference to the GUID.)

dwDataSize
Size, in bytes, of the data chunk.

lpData
Pointer to the constant data.

lpContext
Context passed to the callback function.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

The service provider should examine the GUID in the *guidDataType* parameter and process or store the value specified in *lpData*. Unrecognized values in *guidDataType* can be ignored.

Any pointers returned in a callback function are temporary and are valid only in the body of the callback function. If the application needs to save pointer information, it must allocate memory to hold the data, copy the data, and then store the pointer to this new data. In this function, *lpData* is temporary.

EnumAddressTypeCallback

Application-defined callback function for the IDirectPlayLobby2::EnumAddressTypes method.

BOOL FAR PASCAL EnumAddressTypeCallback(
REFGUID *guidDataType*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

guidDataType
Pointer to a globally unique identifier (GUID) indicating the address type. (In C++, it is a reference to the GUID.) Predefined address types are DPAID_Phone, DPAID_INet, and DPAID_ComPort. For more information about these address types, see DirectPlay Address.

lpContext
Context passed to the callback function.

dwFlags
Reserved; do not use.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

EnumConnectionsCallback

Application-defined callback function for the IDirectPlay3::EnumConnections method.

BOOL FAR PASCAL EnumConnectionsCallback(
LPCGUID *lpguidSP*,
LPVOID *lpConnection*,
DWORD *dwConnectionSize*,
LPCDPNAME *lpName*,
DWORD *dwFlags*,
LPVOID *lpContext*
);

Parameters

lpguidSP

The GUID of the DirectPlay service provider or lobby provider associated with the connection. Use this GUID to uniquely identify the service or lobby provider, rather than using the order in the enumeration or the name.

lpConnection

A read-only pointer to a buffer that contains the connection. This parameter is passed to the IDirectPlay3::InitializeConnection method to initialize the DirectPlay object. This buffer contains a DirectPlay Address.

dwConnectionSize

The size, in bytes, of the *lpConnection* buffer.

lpName

A read-only pointer to a DPNAME structure. The structure contains the short name of the connection that should appear to the user.

If IDirectPlay3::EnumConnections was called on an ANSI interface, reference the strings as ANSI. If EnumConnections was called on a Unicode interface, reference the strings as Unicode.

dwFlags

Flags to indicate the type of connection. Not used at this time.

lpContext

Pointer to an application-defined context.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

The application must implement this function in order to use the IDirectPlay3::EnumConnections method. It is called once for each connection that is enumerated.

The application should allocate memory and copy each of the connections for presentation to the user and for use in the IDirectPlay3::InitializeConnection method.

See Also

IDirectPlay3::EnumConnections, IDirectPlay3::InitializeConnection, Using DirectPlay Addresses

EnumDPCallback

Application-defined callback function for the DirectPlayEnumerate function. Depending on whether UNICODE is defined or not, the prototype for the callback function will have *lpSPName* defined as either the **LPWSTR** type (for Unicode) or the **LPSTR** type (for ANSI).

BOOL FAR PASCAL EnumDPCallback(
LPGUID *lpguidSP*,
LPSTR/LPWSTR *lpSPName*,
DWORD *dwMajorVersion*,
DWORD *dwMinorVersion*,
LPVOID *lpContext*
);

Parameters

lpguidSP
Pointer to the unique identifier of the DirectPlay service provider.

lpSPName
Pointer to a string containing the driver description. Depending on whether the UNICODE symbol is defined or not, the parameter will be of the **LPWSTR** type (Unicode) or the **LPSTR** type (ANSI).

dwMajorVersion and *dwMinorVersion*
Major and minor version numbers of the driver.

lpContext
Pointer to an application-defined context.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

Any pointers returned in a callback function are temporary and are valid only in the body of the callback function. If the application needs to save pointer information, it must allocate memory to hold the data, copy the data, and then store the pointer to this new data. In this function, *lpGUIDSP* and *lpSPName* are temporary.

EnumLocalApplicationsCallback

Application-defined callback function for the IDirectPlayLobby2::EnumLocalApplications method.

BOOL FAR PASCAL EnumLocalApplicationsCallback(
LPCDPLAPPINFO *lpAppInfo*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpAppInfo
Pointer to a read-only DPLAPPINFO structure containing information about the application being enumerated.

lpContext
Context passed from the IDirectPlayLobby2::EnumLocalApplications call.

dwFlags
Reserved; do not use.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

Any pointers returned in a callback function are temporary and are valid only in the body of the callback function. If the application needs to save pointer information, it must allocate memory to hold the data, copy the data, and then store the pointer to this new data. In this function, *lpAppInfo* is temporary. Also note that the pointers inside the structure specified in the *lpAppInfo* parameter — lpszAppNameA and lpszAppName — are also temporary.

EnumPlayersCallback2

Application-defined callback function. The application must implement this function and pass a pointer to it in the IDirectPlay3::EnumGroups, IDirectPlay3::EnumGroupPlayers, IDirectPlay3::EnumPlayers, and IDirectPlay3::EnumGroupsInGroup methods. The callback is called once for each player/group that is enumerated.

BOOL FAR PASCAL EnumPlayersCallback2(
DPID *dpId*,
DWORD *dwPlayerType*,
LPCDPNAME *lpName*,
DWORD *dwFlags*,
LPVOID *lpContext*
);

Parameters

dpId

The DPID of the player or group being enumerated.

dwPlayerType

Type of entity, either DPPLAYERTYPE_GROUP or DPPLAYERTYPE_PLAYER.

lpName

Read only pointer to a DPNAME structure containing the name of the player or group. This pointer is only valid for the duration of the callback function. Any data that is to be saved for future reference must be copied to some application owned memory.

dwFlags

Flags describing the group or player being enumerated.

DPENUMGROUPS_SHORTCUT — the group is a shortcut.

DPENUMGROUPS_STAGINGAREA — the group is a staging area.

DPENUMPLAYERS_GROUP — both players and groups are being enumerated. This flag is returned only if it was specified in the enumeration method calling this callback.

DPENUMPLAYERS_LOCAL — the player or group exists on a local computer. This flag is returned only if it was specified in the enumeration method calling this callback.

DPENUMPLAYERS_REMOTE — the player or group exists on a remote computer. This flag is returned only if it was specified in the enumeration method calling this callback.

DPENUMPLAYERS_SESSION — the player or group exists in the session identified by *lpguidInstance* in the enumeration method. This flag is returned only if it was specified in the enumeration method calling this callback.

DPENUMPLAYERS_SERVERPLAYER — the player is the server player in an application/server session. Only one server player exists in each session.

DPENUMPLAYERS_SPECTATOR — the player is a spectator (applies to players only).

lpContext

Pointer to an application-defined context.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

Any pointers returned in a callback function are temporary and are valid only in the body of the callback function. If the application needs to save pointer information, it must allocate memory to hold the data, copy the data, and then store the pointer to this new data.

See Also

DPNAME, IDirectPlay3::EnumGroups, IDirectPlay3::EnumPlayers, IDirectPlay3::EnumGroupPlayers, IDirectPlay3::EnumGroupsInGroup

EnumSessionsCallback2

Application-defined callback function for the IDirectPlay3::EnumSessions method.

BOOL FAR PASCAL EnumSessionsCallback2(
LPCDPSESSIONDESC2 *lpThisSD*,
LPDWORD *lpdwTimeOut*,
DWORD *dwFlags*,
LPVOID *lpContext*
);

Parameters

lpThisSD

Pointer to a DPSESSIONDESC2 structure describing the enumerated session. This parameter will be set to NULL if the enumeration has timed out.

lpdwTimeOut

Pointer to a variable containing the current time-out value. This parameter can be reset when the DPESC_TIMEDOUT flag is returned if you want to wait longer for sessions to reply.

dwFlags

Typically, this flag is set to zero.

DPESC_TIMEDOUT

The enumeration has timed out. Reset *lpdwTimeOut* and return TRUE to continue, or FALSE to stop the enumeration.

lpContext

Pointer to an application-defined context.

Return Values

Returns TRUE to continue the enumeration or FALSE to stop it.

Remarks

The application must implement this function in order to use the IDirectPlay3::EnumSessions method. This callback function will be called once for each session that is enumerated. Once all the session are enumerated, the callback function will be called one additional time with the DPESC_TIMEDOUT flag.

Applications should look at the flags of the DPSESSIONDESC2 to determine the nature of the session.

Any pointers returned in a callback function are temporary and are valid only in the body of the callback function. If the application needs to save pointer information, it must allocate memory to hold the data, copy the data, and then store the pointer to this new data. In this function, *lpThisSD* is temporary. Also note that the pointers inside the structure specified in the *lpThisSD* parameter — *lpszSessionName* / *lpszSessionNameA* and *lpszPassword* / *lpszPasswordA* — are also temporary.

IDirectPlay3

Applications use the methods of the **IDirectPlay3** interface to create DirectPlay objects and work with system-level variables. (The **IDirectPlay3A** interface is the same as the **IDirectPlay3** interface, except that **IDirectPlay3A** uses ANSI characters, and **IDirectPlay3** uses Unicode.) This section is a reference to the methods of this interface.

The methods of the **IDirectPlay3** interface can be organized into the following groups:

Session Management

Close
EnumConnections
EnumSessions
GetCaps
GetGroupConnectionSettings
GetSessionDesc
InitializeConnection
Open
SetGroupConnectionSettings
SetSessionDesc
StartSession

Player management

CreatePlayer
DestroyPlayer
EnumPlayers
GetPlayerAddress
GetPlayerCaps
GetPlayerData
GetPlayerName
SetPlayerData
SetPlayerName

Message Management

GetMessageCount
Receive
Send
SendChatMessage

Group management

AddGroupToGroup
AddPlayerToGroup
CreateGroup
CreateGroupInGroup
DeleteGroupFromGroup
DeletePlayerFromGroup
DestroyGroup
EnumGroupPlayers
EnumGroups
EnumGroupsInGroup
GetGroupData
GetGroupName
SetGroupData
SetGroupName

Initialization	Initialize
Security and Authentication	SecureOpen

The **IDirectPlay3** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

IDirectPlay3::AddGroupToGroup

Adds a shortcut to a group to an already existing group. This allows the linked group to be enumerated by the IDirectPlay3::EnumGroupsInGroup method. To remove a group's shortcut from another group, call DeleteGroupFromGroup.

HRESULT AddGroupToGroup(
DPID *idParentGroup*,
DPID *idGroup*
);

Parameters

idParentGroup
ID of the group to which the shortcut will be added. Can be any valid group ID.

idGroup
The group ID of the group whose shortcut will be added. Can be any valid group ID.

Return Values

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

DPERR_ACCESSDENIED
DPERR_INVALIDGROUP

Remarks

This method can be used to place a group "inside" more than one group.

A DPMSG_ADDGROUPTOGROUP system message is generated to inform players of this change.

See Also

IDirectPlay3::CreateGroupInGroup, IDirectPlay3::DeleteGroupFromGroup, IDirectPlay3::EnumGroupsInGroup, DPMSG_ADDGROUPTOGROUP

IDirectPlay3::AddPlayerToGroup

Add a player to an existing group. A player can be a member of multiple groups. Groups cannot be added to other groups using this API (see IDirectPlay3::AddGroupToGroup). An application can add any player to any group (including players and groups that weren't created locally).

HRESULT AddPlayerToGroup(
DPID *idGroup*,
DPID *idPlayer*
);

Parameters

idGroup
Group ID of the group to be augmented.

idPlayer
Player ID of the player to be added to the group.

Return Values

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

DPERR_CANTADDPLAYER
DPERR_INVALIDGROUP
DPERR_INVALIDPLAYER
DPERR_NOSESSIONS

This method returns DPERR_INVALIDPLAYER if the player DPID is not a player id or if the id is not for a local player.

Remarks

A DPMSG_ADDPLAYERTOGROUP system message will be generated and sent to all the other players.

See Also

IDirectPlay3::CreateGroup, IDirectPlay3::DeletePlayerFromGroup, DPMSG_ADDPLAYERTOGROUP

IDirectPlay3::Close

Closes a previously opened session. Any locally created groups will migrate to be owned by the host of the session.

HRESULT Close( );

Return Values

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

DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_NOSESSIONS

Remarks

All locally created players will be destroyed and appropriate DPMSG_DELETEPLAYERFROMGROUP and DPMSG_DESTROYPLAYERORGROUP system messages will be sent to other session participants.

See Also

IDirectPlay3::DestroyPlayer, DPMSG_DESTROYPLAYERORGROUP, IDirectPlay3::Open

IDirectPlay3::CreateGroup

Creates a group in the current session. A group is a logical collection of players or other groups.

HRESULT CreateGroup(
LPDPID *lpidGroup*,
LPDPNAME *lpGroupName*,
LPVOID *lpData*,
DWORD *dwDataSize*,
DWORD *dwFlags*
);

Parameters

lpidGroup
Pointer to a variable that will be filled with the DirectPlay group ID. This value is defined by DirectPlay.

lpGroupName
Pointer to a DPNAME structure that holds the name of the group. NULL indicates that the group has no initial name. The name in *lpGroupName* is provided for human use only; it is not used internally and need not be unique.

lpData
Pointer to a block of application-defined remote data to associate initially with the Group ID. NULL indicates that the group has no initial data. The data specified here is assumed to be remote data that will be propagated to all the other applications in the session as if IDirectPlay3::SetGroupData were called.

dwDataSize
Size, in bytes, of the data block that *lpData* points to.

dwFlags
Flag indicating what type of group to create. By default (*dwFlags* = 0), ownership of the group will migrate to the host when the owner leaves the session, and the group persists until it is explicitly destroyed.

DPGROUP_STAGINGAREA – the group is created as a staging area. A staging area is used to marshal players together in order to launch a new session.

Return Values

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

DPERR_CANTADDPLAYER
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS

Remarks

This method will generate a DPMSG_CREATEPLAYERORGROUP system message that will be sent to all the other players. The application can use IDirectPlay3::Send to send a message to all the players in a group by sending one message to the group ID. DirectPlay will either use multicast to send the message (if the service provider supports it) or send individual messages to each player in the group.

The group ID returned to the application should be used to identify the group for message passing and data association. Player and group IDs assigned by DirectPlay will always be unique within the session.

Groups created with **CreateGroup** are top-level groups in the session. They are enumerated with IDirectPlay3::EnumGroups. In contrast, the IDirectPlay3::CreateGroupInGroup method creates a group that is a sub-group of a parent group.

Groups can also be used by the application for general organization. In a lobby session, a staging area

is used as the mechanism for collecting players for the purpose of starting a new application session using IDirectPlay3::StartSession.

The player that creates a group is the default owner of it. Only the owner can change group properties such as the name and remote data. If the owner leaves the session, ownership is transferred to the host of the session.

Any player in the session can change the membership of the group or delete the group.

Groups will persist in the session until they are explicitly destroyed.

See Also

DPNAME, DPMSG_CREATEPLAYERORGROUP, IDirectPlay3::DestroyGroup, IDirectPlay3::EnumGroups, IDirectPlay3::EnumGroupPlayers, IDirectPlay3::Send, IDirectPlay3::SetGroupData, IDirectPlay3::SetGroupName, IDirectPlay3::CreateGroupInGroup, IDirectPlay3::GetGroupFlags

IDirectPlay3::CreateGroupInGroup

Creates a group within an existing group. A group created within another group can only be enumerated using the IDirectPlay3::EnumGroupsInGroup method. A group created this way can be destroyed by calling the IDirectPlay3::DestroyGroup method.

HRESULT CreateGroupInGroup(
DPID *idParentGroup*,
LPDPID *lpidGroup*,
LPDPNAME *lpGroupName*,
LPVOID *lpData*,
DWORD *dwDataSize*,
DWORD *dwFlags*
);

Parameters

idParentGroup
The DPID of the group within which a group will be created. Must be an already existing group.

lpidGroup
Pointer to the DPID that will be filled in with the DirectPlay group ID of the created group.

lpGroupName
Pointer to a DPNAME structure that holds the name of the group to be created. NULL indicates that the group has no initial name.

lpData
Pointer to a block of application-defined remote data to associate initially with the group ID. NULL indicates that the group has no initial data. The data specified here is assumed to be remote data that will be propagated to all the other applications in the session as if the IDirectPlay3::SetGroupData method had been called.

dwDataSize
Size, in bytes, of the data block that the *lpData* parameter points to.

dwFlags
Flag indicating what type of group to create. Default (*dwFlags* = 0) is a normal group.
DPGROUP_STAGINGAREA – the group can be used to launch DirectPlay sessions using the IDirectPlay3::StartSession method.

Return Values

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

DPERR_CANTADDPLAYER
DPERR_INVALIDFLAGS
DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS

This method returns DPERR_CANTADDPLAYER if the group could not be created. It returns DPERR_INVALIDGROUP if the parent group ID is invalid.

Remarks

A DPMSG_CREATEPLAYERORGROUP system message is generated to inform players of this change.

See Also

IDirectPlay3::DestroyGroup, IDirectPlay3::EnumGroupsInGroup, DPMSG_CREATEPLAYERORGROUP

IDirectPlay3::CreatePlayer

Creates a local player for the current session.

HRESULT CreatePlayer(
LPDPID *lpidPlayer*,
LPDPNAME *lpPlayerName*,
HANDLE *hEvent*,
LPVOID *lpData*,
DWORD *dwDataSize*,
DWORD *dwFlags*
);

Parameters

lpidPlayer

Pointer to a variable that will be filled with the DirectPlay player ID. This value is defined by DirectPlay.

lpPlayerName

Pointer to a DPNAME structure that holds the name of the player. NULL indicates that the player has no initial name information. The name in *lpPlayerName* is provided for human use only. It is not used internally and need not be unique.

hEvent

An event object created by the application that will be signaled by DirectPlay when a message addressed to this player is received.

lpData

Pointer to a block of application-defined data to associate with the player ID. NULL indicates that the player has no initial data. The data specified in this parameter is assumed to be remote data that will be propagated to all the other applications in the session, as if IDirectPlay3::SetPlayerData were called.

dwDataSize

Size, in bytes, of the data block that *lpData* points to.

dwFlags

Flags indicating what type of player this is. Default (*dwFlags* = 0) is a nonspectator, nonserver player.

DPPLAYER_SERVERPLAYER – the player is a server player for client/server communications. Only the host can create a server player. There can only be one server player in a session. **CreatePlayer** will always return a player ID of DPID_SERVERPLAYER if this flag is specified.

DPPLAYER_SPECTATOR – the player is created as a spectator. A spectator player behaves exactly as a normal player except the player is flagged as a spectator. The application can then limit what a spectator player can do. The behavior of a spectator player is completely defined by the application. DirectPlay simply propagates this flag.

Return Values

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

DPERR_CANTADDPLAYER
DPERR_CANTCREATEPLAYER
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_NOCONNECTION

Remarks

A single process can have multiple local players that communicate through a DirectPlay object with other players on the same computer or players on remote computers.

Upon successful completion, this method sends a DPMSG_CREATEPLAYERORGROUP system message to all the other players in the session announcing that a new player has joined the session. By default, all local players receive copies of all the system messages.

Your application should use the player ID returned to the application to identify the player for message passing and data association. Player and group IDs assigned by DirectPlay will always be unique within the session.

If the application closes the session, any local players created will be automatically destroyed. Only the application that created the player can:

- destroy the player.
- change the player's name or remote data.
- send messages from the player.

If the application uses a separate thread to retrieve DirectPlay messages, it is highly recommended that a non-NULL *hEvent* be supplied and used for synchronization. This event will be set when this player receives a message. Within the message receive thread, use the Win32 API **WaitForSingleObject** (or use **WaitForMultipleObjects** if more than one event is used) within the thread to determine if a player has messages. It is inefficient to loop on IDirectPlay3::Receive inside a separate thread waiting for a message. The same event can be used for all the local players, or the application can supply different events for each player. The application is responsible for creating and destroying the event. See Synchronization for more information.

See Also

DPNAME, DPMSG_CREATEPLAYERORGROUP, IDirectPlay3::DestroyPlayer, IDirectPlay3::EnumPlayers, IDirectPlay3::Receive, IDirectPlay3::Send, IDirectPlay3::SetPlayerData, IDirectPlay3::SetPlayerName, IDirectPlay3::GetPlayerFlags

IDirectPlay3::DeleteGroupFromGroup

Removes a shortcut to a group previously added with the IDirectPlay3::AddGroupToGroup method from a group. Deleting the shortcut does not destroy the group.

HRESULT DeleteGroupFromGroup(
DPID *idParentGroup*,
DPID *idGroup*
);

Parameters

idParentGroup
The group DPID of the group containing the shortcut to be deleted. Can be any valid group DPID.

idGroup
The group DPID of the group to delete. Can be any valid group DPID.

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_ACCESSDENIED
DPERR_INVALIDGROUP

Remarks

A DPMSG_DELETEGROUPFROMGROUP system message is generated to inform players of this change.

See Also

IDirectPlay3::AddGroupToGroup, DPMSG_DELETEGROUPFROMGROUP

IDirectPlay3::DeletePlayerFromGroup

Removes a player from a group.

HRESULT DeletePlayerFromGroup(
DPID *idGroup*,
DPID *idPlayer*
);

Parameters

idGroup
Group ID of the group to be adjusted.

idPlayer
Player ID of the player to be removed from the group.

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER

Remarks

A DPSYS_DELETEPLAYERFROMGROUP system message is generated to inform the other players of this change. For a list of system messages, see Using System Messages.

A player can delete any player from a group, even if that group was created by some other computer.

See Also

IDirectPlay3::AddPlayerToGroup, DPMSG_DELETEPLAYERFROMGROUP

IDirectPlay3::DestroyGroup

Deletes a group from the session. The ID belonging to this group will not be reused during the current session.

HRESULT DestroyGroup(
DPID *idGroup*
);

Parameters

idGroup
The ID of the group being removed from the game.

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

It is not necessary to empty a group before deleting it. The individual players belonging to the group are not destroyed. This method will generate a DPMSG_DELETEPLAYERFROMGROUP system message for each player in the group, and then a DPMSG_DESTROYPLAYERORGROUP system message. For a list of system messages, see Using System Messages.

Any application can destroy any group even if the group was not created locally.

See Also

IDirectPlay3::CreateGroup, DPMSG_DESTROYPLAYERORGROUP

IDirectPlay3::DestroyPlayer

Deletes a local player from the 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.

HRESULT DestroyPlayer(
DPID *idPlayer*
);

Parameters

idPlayer
Player ID of the player that is being removed from the session.

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_ACCESSDENIED
DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER

Remarks

This method will generate a DPMSG_DELETEPLAYERFROMGROUP system message for each group that the player belongs to, and then a DPMSG_DESTROYPLAYERORGROUP system message.

Only the application that created the player can destroy it.

See Also

IDirectPlay3::CreatePlayer, DPMSG_DESTROYPLAYERORGROUP

IDirectPlay3::EnumConnections

Enumerates all the registered service providers and lobby providers that are available to the application. These should be presented to the user to make a selection. The connection that the user selects should be passed to the IDirectPlay3::InitializeConnection method.

HRESULT EnumConnections(
LPCGUID *lpguidApplication*,
LPDPENUMCONNECTIONSCALLBACK *lpEnumCallback*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpguidApplication

Pointer to an application GUID. Only service providers and lobby providers that are usable by this application will be returned. If set to a NULL pointer, a list of all the connections is enumerated regardless of the application GUID.

lpEnumCallback

Pointer to a user-supplied EnumConnectionsCallback function that will be called for each available connection.

lpContext

Pointer to a user-defined context that is passed to the callback function.

dwFlags

Flags that specify the type of connections to be enumerated. The default (*dwFlags* = 0) will enumerate DirectPlay service providers only. Possible values are:

DPCONNECTION_DIRECTPLAY – enumerate DirectPlay service providers to communicate in an application session.

DPCONNECTION_DIRECTPLAYLOBBY – enumerate DirectPlay lobby providers to communicate with a lobby server.

Return Values

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

DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS

Remarks

This method replaces the DirectPlayEnumerate function. DirectPlayEnumerate still works, but only returns registered service providers.

The order in which the service and lobby providers are returned is not guaranteed to be the same in subsequent calls to **EnumConnections**

Not all the enumerated connections are available for use. For example, this method will return the Modem service provider even if the user has no modem installed. The application can call the IDirectPlay3::InitializeConnection method on each connection and check for an error code to determine if the service provider can be used.

See Also

IDirectPlay3::InitializeConnection, EnumConnectionsCallback, DirectPlay Address

IDirectPlay3::EnumGroupPlayers

Enumerates the players belonging to a specific group in the currently open session. If there is no open session, players in a remote session can be enumerated by specifying the DPENUMPLAYERS_SESSION flag and the *guidInstance* of the session. Password protected remote sessions cannot be enumerated.

A pointer to an application-implemented callback function must be supplied, and DirectPlay calls it once for each player in the group that matches the criteria specified in *dwFlags*.

You can't use **EnumGroupPlayers** in a lobby session you're not connected to.

HRESULT EnumGroupPlayers(
DPID *idGroup*,
LPGUID *lpguidInstance*,
LPDPENUMPLAYERSCALLBACK2 *lpEnumPlayersCallback2*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

idGroup
DPID of the group whose players are to be enumerated.

lpguidInstance
Pointer to a GUID identifying the session to be enumerated. This parameter is ignored unless the DPENUMPLAYERS_SESSION flag is specified. The GUID must be equal to one of the sessions enumerated by IDirectPlay3::EnumSessions.

lpEnumPlayersCallback2
Pointer to the EnumPlayersCallback2 function that will be called for every player in the group that matches the criteria specified in *dwFlags*.

lpContext
Pointer to an application-defined context that is passed to each enumeration callback.

dwFlags
Flags specifying how the enumeration is to be done. The default (*dwFlags* = 0) enumerates players in a group in the current active session. You should OR together the flags that you want to specify. Only players that meet all the criteria of the combined flags will be enumerated. For example, if you specify (DPENUMPLAYERS_LOCAL | DPENUMPLAYERS_SPECTATOR) only players in the group that are local and are spectator players will be enumerated. If you specify (DPENUMPLAYERS_LOCAL | DPENUMPLAYERS_REMOTE), no players will be enumerated because no player can be both local and remote.

Can be one or more of the following values:

DPENUMPLAYERS_LOCAL
Enumerates players created locally by this DirectPlay object.

DPENUMPLAYERS_REMOTE
Enumerates players created by remote DirectPlay objects.

DPENUMPLAYERS_SERVERPLAYER
Enumerates the server player.

DPENUMPLAYERS_SESSION
Enumerates the players for the session identified by *lpguidInstance*. This flag can only be used if there is no current open session. This flag can't be used in a lobby session.

DPENUMPLAYERS_SPECTATOR
Enumerates players who are spectator players.

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_ACCESSDENIED
DPERR_INVALIDPARAMS
DPERR_INVALIDGROUP
DPERR_NOSESSIONS
DPERR_UNAVAILABLE

This method returns DPERR_ACCESSDENIED if the session is a lobby session you're not connected to. It returns DPERR_INVALIDPARAMS if an invalid callback, an invalid *guidInstance*, or invalid flags were supplied. It returns DPERR_NOSESSIONS if there is no active session. It returns DPERR_UNAVAILABLE if the session could not be enumerated.

Remarks

By default, this method will enumerate all players in the current session. The DPENUMPLAYERS_SESSION flag can be used, along with a session instance GUID, to request that a session's host provide its list for enumeration. Use of the DPENUMPLAYERS_SESSION flag with this method must occur after the IDirectPlay3::EnumSessions method has been called, and before any calls to the IDirectPlay3::Close or IDirectPlay3::Open methods.

See Also

IDirectPlay3::CreatePlayer, IDirectPlay3::DestroyPlayer, IDirectPlay3::AddPlayerToGroup, IDirectPlay3::DeletePlayerFromGroup

IDirectPlay3::EnumGroups

Enumerates all the top-level groups in the current session. Top-level groups are groups that were created with the IDirectPlay3::CreateGroup method. If there is no open session, groups in a remote session can be enumerated by specifying the DPENUMGROUPS_SESSION flag and the *guidInstance* of the session. Password protected remote sessions cannot be enumerated.

A pointer to an application-implemented callback function must be supplied, and DirectPlay calls it once for each group in the session that matches the criteria specified in *dwFlags*.

You can't use **EnumGroups** in a lobby session you're not connected to.

HRESULT EnumGroups(
LPGUID *lpguidInstance*,
LPDPENUMPLAYERSCALLBACK2 *lpEnumPlayersCallback2*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpguidInstance

This parameter should be NULL to enumerate groups in the currently open session. If there is no open session, this parameter can be a pointer to a GUID identifying the session to be enumerated. The GUID must be equal to one of the sessions enumerated by IDirectPlay3::EnumSessions. This parameter is ignored unless the DPENUMGROUPS_SESSION flag is specified.

lpEnumPlayersCallback2

Pointer to the EnumPlayersCallback2 function that will be called for every group in the session that matches the criteria specified in *dwFlags*.

lpContext

Pointer to an application-defined context that is passed to each enumeration callback.

dwFlags

Flags specifying how the enumeration is to be done. Default (dwFlags = 0) enumerates all groups in the current active session. You should OR together the flags that you want to specify. Only groups that meet all the criteria of the combined flags will be enumerated. For example, if you specify (DPENUMGROUPS_LOCAL | DPENUMGROUPS_STAGINGAREA) only groups that are local and are staging areas will be enumerated. If you specify (DPENUMGROUPS_LOCAL | DPENUMGROUPS_REMOTE), no groups will be enumerated because no group can be both local and remote.

Can be one or more of the following values:

DPENUMGROUPS_ALL

Enumerates all groups in this session.

DPENUMGROUPS_LOCAL

Enumerates groups created locally by this DirectPlay object.

DPENUMGROUPS_REMOTE

Enumerates groups created by remote DirectPlay objects.

DPENUMGROUPS_SESSION

Performs enumeration in the session identified by the *lpguidInstance* parameter. This flag can only be used if there is no current active session. You can't use this flag in a lobby session.

DPENUMGROUPS_STAGINGAREA

Enumerates groups that are staging areas. (Staging areas are used to marshal players together in order to launch a new session.)

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_INVALIDPARAMS
DPERR_NOSESSIONS
DPERR_UNAVAILABLE

This method returns DPERR_INVALIDPARAMS if an invalid callback, an invalid *guidInstance*, or invalid flags were supplied. It returns DPERR_NOSESSIONS if there is no open session. It returns DPERR_UNAVAILABLE if the remote session could not be enumerated.

Remarks

To enumerate the subgroups in a group, use IDirectPlay3::EnumGroupsInGroup.

See Also

EnumPlayersCallback2, IDirectPlay3::CreateGroup, IDirectPlay3::DestroyGroup, IDirectPlay3::EnumSessions

IDirectPlay3::EnumGroupsInGroup

Enumerates all groups and shortcuts to groups that are contained within another group. Groups are placed inside other groups by creating them with the IDirectPlay3::CreateGroupInGroup method or by adding them to a group with the IDirectPlay3::AddGroupToGroup method. This method is not recursive.

You can't use **EnumGroupsInGroup** in a lobby session you're not connected to.

HRESULT EnumGroupsInGroup(
DPID *idGroup*,
LPGUID *lpguidInstance*,
LPDPENUMPLAYERSCALLBACK2 *lpEnumCallback*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

idGroup

DPID of the group whose subgroups are to be enumerated.

lpguidInstance

Pointer to a GUID identifying the session to be enumerated. This parameter is ignored unless the DPENUMPLAYERS_SESSION flag is specified. The GUID must equal one of the sessions enumerated by the IDirectPlay3::EnumSessions method.

lpEnumCallback

Pointer to the EnumPlayersCallback2 function that will be called for every group in the group that matches the criteria specified in *dwFlags*.

lpContext

Pointer to an application-defined context that is passed to each enumeration callback.

dwFlags

Flags specifying how the enumeration will be done. The default (*dwFlags* = 0) enumerates all groups in the current active session. You should OR together the flags that you want to specify. Only groups that meet all the criteria of the combined flags will be enumerated. For example, if you specify (DPENUMGROUPS_LOCAL | DPENUMGROUPS_STAGINGAREA) only groups that are local and are staging areas will be enumerated. If you specify (DPENUMGROUPS_LOCAL | DPENUMGROUPS_REMOTE), no groups will be enumerated because no group can be both local and remote.

Can be one or more of the following values:

DPENUMGROUPS_ALL

Enumerates all groups in the group.

DPENUMGROUPS_LOCAL

Enumerates groups in the group created locally by this DirectPlay object.

DPENUMGROUPS_REMOTE

Enumerates groups in the group created by remote DirectPlay objects.

DPENUMGROUPS_SESSION

Performs enumeration in the session identified by the *lpguidInstance* parameter. This flag can only be used if there is no current active session. This flag can't be used in lobby sessions.

DPENUMGROUPS_SHORTCUT

Enumerates groups that are shortcuts added to the group using IDirectPlay3::AddGroupToGroup. (A shortcut is a link to another group.)

DPENUMGROUPS_STAGINGAREA

Enumerates groups in the group that are staging areas. (Staging areas are used to marshal players together in order to launch a new session.)

Return Values

Returns DP_OK if successful, or one of the following error messages otherwise:

DPERR_INVALIDFLAGS
DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS
DPERR_NOSESSIONS
DPERR_UNSUPPORTED

This method returns DPERR_INVALIDPARAMS if an invalid callback or an invalid *guidInstance* is supplied. It returns DPERR_NOSESSIONS if there is no active session. It returns DPERR_UNSUPPORTED if the session could not be enumerated.

See Also

IDirectPlay3::CreateGroupInGroup, IDirectPlay3::DestroyGroup, IDirectPlay3::AddGroupToGroup, IDirectPlay3::DeleteGroupFromGroup

IDirectPlay3::EnumPlayers

Enumerates the players in the current open session. If there is no open session, players in a remote session can be enumerated by specifying the DPENUMPLAYERS_SESSION flag and the *guidInstance* of the session. Password protected remote sessions cannot be enumerated.

A pointer to an application-implemented callback function must be supplied and DirectPlay calls it once for each player in the session that matches the criteria specified in *dwFlags*.

Within a lobby session, this method will always return DPERR_ACCESSDENIED.

HRESULT EnumPlayers(
LPGUID *lpguidInstance*,
LPDPENUMPLAYERSCALLBACK2 *lpEnumPlayersCallback2*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpguidInstance

This parameter should be NULL to enumerate players in the currently open session. If there is no open session, this parameter can be a pointer to a GUID identifying the session to be enumerated. The GUID must be equal to one of the sessions enumerated by IDirectPlay3::EnumSessions. This parameter is ignored unless the DPENUMPLAYERS_SESSION flag is specified.

lpEnumPlayersCallback2

Pointer to the EnumPlayersCallback2 function that will be called for every player in the session that matches the criteria specified in *dwFlags*.

lpContext

Pointer to an application-defined context that is passed to each enumeration callback.

dwFlags

Flags specifying how the enumeration is to be done. Default (*dwFlags* = 0) enumerates all players in the current active session. You should OR together the flags that you want to specify. Only players that meet all the criteria of the combined flags will be enumerated. For example, if you specify (DPENUMPLAYERS_LOCAL | DPENUMPLAYERS_SPECTATOR) only players that are local and are spectator players will be enumerated. If you specify (DPENUMPLAYERS_LOCAL | DPENUMPLAYERS_REMOTE), no players will be enumerated because no player can be both local and remote.

Can be one or more of the following values:

DPENUMPLAYERS_ALL

Enumerates all players in this session.

DPENUMPLAYERS_GROUP

Includes groups in the enumeration of players.

DPENUMPLAYERS_LOCAL

Enumerates players created locally by this DirectPlay object.

DPENUMPLAYERS_REMOTE

Enumerates players created by remote DirectPlay objects.

DPENUMPLAYERS_SERVERPLAYER

Enumerates the server player.

DPENUMPLAYERS_SESSION

Enumerates the players for the session identified by *lpguidInstance*. This flag can only be used if there is no current open session.

DPENUMPLAYERS_SPECTATOR

Enumerates players who are spectator players.

Return Values

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

DPERR_INVALIDPARAMS
DPERR_NOSESSIONS
DPERR_UNAVAILABLE

This method returns DPERR_INVALIDPARAMS if an invalid callback, an invalid *guidInstance*, or invalid flags were supplied. It returns DPERR_NOSESSIONS if there is no open session. It returns DPERR_UNAVAILABLE if the remote session could not be enumerated.

Remarks

By default, this method will enumerate players in the current open session. Groups can also be included in the enumeration by using the DPENUMPLAYERS_GROUP flag. The DPENUMPLAYERS_SESSION flag can be used, along with a session instance GUID, to request that a session's host provide its list for enumeration. This method cannot be called from within an IDirectPlay3::EnumSessions enumeration. Furthermore, use of the DPENUMPLAYERS_SESSION flag with this method must occur after the IDirectPlay3::EnumSessions method has been called, and before any calls to the IDirectPlay3::Close or IDirectPlay3::Open methods.

See Also

IDirectPlay3::CreatePlayer, IDirectPlay3::DestroyPlayer, IDirectPlay3::EnumSessions

IDirectPlay3::EnumSessions

Enumerates all the active sessions for a particular application and/or all the active lobby sessions that serve a particular application. This method can be called after a DirectPlay object has been created and initialized either by DirectPlayCreate or IDirectPlay3::InitializeConnection.

This method returns an error if called while a session is already open.

HRESULT EnumSessions(
LPDPSESSIONDESC2 *lpsd*,
DWORD *dwTimeout*,
LPDPENUMSESSIONSCALLBACK2 *lpEnumSessionsCallback2*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpsd

Pointer to the DPSESSIONDESC2 structure describing the sessions to be enumerated. Only those sessions that meet the criteria set in this structure will be enumerated. The *guidApplication* member can be set to the globally unique identifier (GUID) of an application of interest if it is known, or to GUID_NULL to obtain all sessions. The *lpszPassword* member is only needed if you want private sessions. All data members besides *guidApplication* and *lpszPassword* are ignored.

dwTimeout

In the synchronous case, the total amount of time, in milliseconds, that DirectPlay will wait for replies to the enumeration request (not the time between each enumeration). Any replies received after this time-out will be ignored. The application will be blocked until the time-out expires.

In the asynchronous case, this is the interval, in milliseconds, that enumeration requests will be broadcast by DirectPlay in order to update the internal sessions list.

If the time-out is set to zero, a default time-out appropriate for the service provider and connection type will be used. It is recommended that you set this value to zero. The application can determine this time-out by calling IDirectPlay3::GetCaps and examining the *dwTimeout* data member of the DPCAPS structure.

lpEnumSessionsCallback2

Pointer to the application-supplied EnumSessionsCallback2 function to be called for each DirectPlay session responding.

lpContext

Pointer to a user-defined context that is passed to each enumeration callback.

dwFlags

The default is 0, which is equivalent to DPENUMSESSIONS_AVAILABLE. When enumerating sessions with DPENUMSESSIONS_ALL or DPENUMSESSIONS_PASSWORDREQUIRED it is important for the application to know which sessions cannot be joined and which sessions are password-protected so the user can be warned or prompted for a password.

DPENUMSESSIONS_ALL

Enumerate all active sessions, whether they are accepting new players or not. Sessions in which the player limit has been reached, new players have been disabled, or joining has been disabled will be enumerated.

Password protected sessions will not be enumerated unless the DPENUMSESSIONS_PASSWORDREQUIRED flag is also specified.

If DPENUMSESSIONS_ALL is not specified, DPENUMSESSIONS_AVAILABLE is assumed.

DPENUMSESSIONS_ASYNC

Enumerates all the current sessions in the session cache and returns immediately. Starts the asynchronous enumeration process if not already started. Updates to the session list continue until canceled by calling **EnumSessions** with the **DPENUMSESSIONS_STOPASYNC** flag, or by calling Open, or by calling Release.

If this flag is not specified, the enumeration is done synchronously.

DPENUMSESSIONS_AVAILABLE

Enumerate all sessions that are accepting new players to join. Sessions which have reached their maximum number of players, or have disabled new players, or have disabled joining will not be enumerated.

Password protected sessions will not be enumerated unless the **DPENUMSESSIONS_PASSWORDREQUIRED** flag is also specified.

DPENUMSESSIONS_PASSWORDREQUIRED

When used in combination with one of the two flags **DPENUMSESSIONS_AVAILABLE** or **DPENUMSESSIONS_ALL**, enables password-protected sessions to be enumerated in addition to sessions without password protection.

If this flag is not specified, no password protected sessions will be returned.

DPENUMSESSIONS_RETURNSTATUS

If this flag is specified, the enumeration will not display any dialog boxes showing the connection progress status. If the connection cannot be made immediately, the method will return with the DPERR_CONNECTING error. The application must keep calling **EnumSessions** until either DP_OK is returned, indicating successful completion, or some other error code is returned, indicating an error.

DPENUMSESSIONS_STOPASYNC

Enumerates all the current sessions in the session cache and cancels the asynchronous enumeration process.

Return Values

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

DPERR_CONNECTING
DPERR_EXCEPTION
DPERR_GENERIC
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_UNINITIALIZED
DPERR_USERCANCEL

This method returns DPERR_GENERIC if the session is already open. It returns DPERR_UNINITIALIZED if the DirectPlay object has not been initialized. It returns DPERR_USERCANCEL if the user canceled the enumeration process (usually by canceling a service provider dialog box). It returns DPERR_CONNECTING if the method is in the process of connecting to the network.

Remarks

EnumSessions works by requesting that the service provider locate one or more hosts on the network and send them an enumeration request. The replies that are received make up the sessions that are enumerated.

EnumSessions can be called synchronously (default) or asynchronously. When called synchronously, DirectPlay will clear the internal session cache, send out an enumeration request, and collect replies until the specified time-out expires. Each session will then be returned to the application through the callback function. The application will be blocked until all the sessions have been returned through the callback function.

When called asynchronously (**DPENUMSESSIONS_ASYNC**), all the current sessions (if any) in the session cache will be returned to the application through the callback function and then the method will return. DirectPlay will then automatically send out enumeration requests with the period of the time-out parameter and listen for replies. Each reply will be used to update the session cache:

- Sessions already in the cache will be updated
- Sessions that haven't been updated for a set period of time (and thus have expired) will be deleted

- New sessions will be added

The application must call **EnumSessions** (with the **DPENUMSESSIONS_ASYNC** flag) again to obtain the most up-to-date session list with all the expired sessions deleted, new sessions added, and updated sessions. Subsequent calls to **EnumSessions** will not generate an enumeration request. Enumeration requests will be generated periodically by DirectPlay until the process is either canceled by calling **EnumSessions** with the **DPENUMSESSIONS_STOPASYNC** flag, a session is opened using the IDirectPlay3::Open method, or the DirectPlay object is released.

Once enumeration of the available sessions is complete, the application can join one of the sessions using the IDirectPlay3::Open method. Only sessions in the session cache can be opened. It is possible for the application to attempt to open a session that has expired since the last time **EnumSessions** was called in which case an error will be returned.

No authentication is performed when enumerating sessions on a secure server. All sessions that meet the enumeration criteria will be returned. Authentication will be done when the application attempts to open one of these sessions with IDirectPlay3::Open or IDirectPlay3::SecureOpen.

If the application was not started by a lobby, the service provider can display a dialog box asking the user for information (such as a phone number or IP address) in order to perform the enumeration on the network, if that information was not provided in IDirectPlay3::InitializeConnection. If the service provider can do a broadcast enumeration and requires no extra information from the user, no dialog box will appear. If the user cancels a service provider dialog box, **EnumSessions** returns DPERR_USERCANCEL.

See Also

DPSESSIONDESC2, IDirectPlay3::Open, IDirectPlay3::SecureOpen

IDirectPlay3::GetCaps

Obtains the capabilities of this DirectPlay object.

HRESULT GetCaps(
LPDPCAPS *lpDPCaps*,
DWORD *dwFlags*
);

Parameters

lpDPCaps
Pointer to a DPCAPS structure that will be filled with the capabilities of the DirectPlay object. The *dwSize* member of the DPCAPS structure must be filled in before using **IDirectPlay3::GetCaps**.

dwFlags
If this parameter is set to 0, the capabilities will be computed for nonguaranteed messaging.

DPGETCAPS_GUARANTEED
Retrieves the capabilities for a guaranteed message delivery.

Return Values

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

DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS

Remarks

This method returns the capabilities of the current session, while the IDirectPlay3::GetPlayerCaps method returns the capabilities of the requested player.

See Also

DPCAPS, IDirectPlay3::GetPlayerCaps, IDirectPlay3::Send

IDirectPlay3::GetGroupConnectionSettings

Retrieves the connection settings for a group from the DPLCONNECTION structure. Any sessions launched from this group will use these settings. This method can only be used in a lobby session.

HRESULT GetGroupConnectionSettings(
DWORD *dwFlags*,
DPID *idGroup*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

dwFlags
Not used. Must be zero.

idGroup
The DPID of the group to get the connection settings for.

lpData
Pointer to a buffer into which the DPLCONNECTION structure and all its data will be copied. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the minimum size required to hold the data.

lpdwDataSize
Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the minimum buffer size required.

Return Values

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

DPERR_ACCESSDENIED
DPERR_BUFFERTOOSMALL
DPERR_INVALIDGROUP
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_UNSUPPORTED

Remarks

Group connection settings are only relevant for staging area groups.

You can see if a game has been launched by looking at the *guidInstance* data member of the DPSESSIONDESC2 structure, whose pointer is contained in the *lpSessionDesc* data member of the DPLCONNECTION structure returned by this method. The *guidInstance* will be GUID_NULL until a game has been launched.

See Also

DPLCONNECTION, IDirectPlayLobby2::RunApplication, IDirectPlay3::SetGroupConnectionSettings, DPGROUP_STAGINGAREA

IDirectPlay3::GetGroupData

Retrieves an application-specific data block that was associated with a group ID by using IDirectPlay3::SetGroupData.

HRESULT GetGroupData(
DPID *idGroup*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*,
DWORD *dwFlags*
);

Parameters

idGroup
Group ID for which data is being requested.

lpData
Pointer to a buffer where the application-specific group data is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data.

lpdwDataSize
Pointer to a variable that is initialized to the size of the buffer before calling the method. After the method returns, this parameter will be set to the size, in bytes, of the group data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size required.

dwFlags
If this parameter is set to 0, the remote data will be retrieved.

DPGET_LOCAL
Retrieves the local data set by this application

DPGET_REMOTE
Retrieves the current data from the remote shared data space.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

DirectPlay can maintain two types of group data: local and remote. The application must specify which type of data to retrieve. Local data was set by this DirectPlay object by using the DPSET_LOCAL flag. Remote data might have been set by any application in the session by using the DPSET_REMOTE flag.

See Also

IDirectPlay3::SetGroupData

IDirectPlay3::GetGroupFlags

Returns the flags describing the group.

HRESULT GetGroupFlags(
DPID *idGroup*,
LPDWORD *lpdwFlags*
);

Parameters

idGroup

The DPID of the group whose flag settings are to be retrieved.

lpdwFlags

Pointer to a DWORD to be set to the group flag settings. Can be one or more of the following:

DPGROUP_LOCAL – the group was created by this application. If this flag is not specified, the group is a remote group.

DPGROUP_STAGINGAREA – the group is a staging area.

Return Values

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

DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS

See Also

IDirectPlay3::CreateGroup

IDirectPlay3::GetGroupName

Returns the name associated with a group.

HRESULT GetGroupName(
DPID *idGroup*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

idGroup
ID of the group whose name is being requested.

lpData
Pointer to a buffer where the name data is to be written. Set this parameter to NULL to request only the size of data. *lpdwDataSize* will be set to the size required to hold the data.

lpdwDataSize
Pointer to a variable that is initialized to the size of the buffer before calling the method. After the method returns, this parameter will be set to the size, in bytes, of the name data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size that is required.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

After the method returns, the pointer *lpData* should be cast to the DPNAME structure to read the group name data.

See Also

DPNAME, IDirectPlay3::SetGroupName

IDirectPlay3::GetGroupParent

Returns the DPID of the parent of the group.

HRESULT GetGroupParent(
DPID *idGroup***,**
LPDPID *lpidParent*
);

Parameters

idGroup
ID of the group whose parent is being requested.

lpidParent
Pointer to a DPID to be set to the ID of the parent group. If this method returns a parent ID of zero, the group is a root group.

Return Values

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

DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS

See Also

IDirectPlay3::CreateGroupInGroup

IDirectPlay3::GetMessageCount

Queries for the number of messages in the receive queue for a specific local player.

HRESULT GetMessageCount(
DPID *idPlayer*,
LPDWORD *lpdwCount*
);

Parameters

idPlayer
ID of the player whose message count is requested. The player must be local.
lpdwCount
Pointer to a variable that will be set to the message count when this method returns.

Return Values

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

DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

See Also

IDirectPlay3::Receive

IDirectPlay3::GetPlayerAccount

In a secure session, can be called by the session host to obtain account information about the specified player.

HRESULT GetPlayerAccount(
DPID *idPlayer*,
DWORD *dwFlags*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

idPlayer
The DPID of the player whose account information is to be retrieved.

dwFlags
Not used. Must be zero.

lpData
A pointer to a buffer (allocated by the application) where the account data is to be written. A DPACCOUNTDESC structure will be copied as well as any data referenced by the members of the structure; therefore, the number of bytes copied is variable. Cast the *lpData* parameter to LPDPACCOUNTDESC in order to read the name. By passing NULL, the application can request the number of bytes required be put in the *lpdwDataSize* parameter. The application can then allocate the space and call this method again.

lpdwDataSize
A pointer to a DWORD that is initialized with the size of the buffer. After the method returns, *lpdwDataSize* will be set to the number of bytes that were actually copied into the buffer. If the buffer was too small the method returns DPERR_BUFFERTOOSMALL, and *lpdwDataSize* is set to the minimum required buffer size.

Return Values

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

DPERR_ACCESSDENIED
DPERR_BUFFERTOOSMALL
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

The information returned by **GetPlayerAccount** uniquely identifies an account. This information can be used to record the transactions or other activities of a logged-in player.

See Also

IDirectPlay3::SecureOpen

IDirectPlay3::GetPlayerAddress

Retrieves the DirectPlay Address for a player.

The DirectPlay Address is a network address for a player using a specific service provider. You should call the IDirectPlayLobby2::EnumAddress method to parse the DirectPlay Address buffer retrieved by **IDirectPlay3::GetPlayerAddress**.

HRESULT GetPlayerAddress(
DPID *idPlayer*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

idPlayer

Player ID that the address is being requested for. Pass in zero to obtain a list of valid address options for a service provider (for example, a list of valid modem choices for the modem-to-modem service provider).

lpData

Pointer to a buffer where the DirectPlay Address is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data.

lpdwDataSize

Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the group data. If the buffer was too small, then this parameter will be set to the buffer size that is required and the method will return DPERR_BUFFERTOOSMALL.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

To get a list of valid modem choices, pass in zero for the *idPlayer* parameter. A list of modem choices will be returned as a list of ANSI or Unicode strings with a zero-length string at the end.

For more information about the DirectPlay Address, DirectPlay Address.

See Also

IDirectPlayLobby2::EnumAddress

IDirectPlay3::GetPlayerCaps

Retrieves the current capabilities of a specified player.

HRESULT GetPlayerCaps(
DPID *idPlayer*,
LPDPCAPS *lpPlayerCaps*,
DWORD *dwFlags*
);

Parameters

idPlayer
Player ID for which the capabilities should be computed.

lpPlayerCaps
Pointer to a DPCAPS structure that will be filled with the capabilities. The *dwSize* member of the DPCAPS structure must be filled in before using **IDirectPlay3::GetPlayerCaps**.

dwFlags
If this parameter is set to 0, the capabilities will be computed for nonguaranteed messaging.

DPGETCAPS_GUARANTEED
Retrieves the capabilities for a guaranteed message delivery.

Return Values

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

DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

This method returns the capabilities of the requested player, while the IDirectPlay3::GetCaps method returns the capabilities of the current session.

See Also

DPCAPS, IDirectPlay3::GetCaps, IDirectPlay3::Send

IDirectPlay3::GetPlayerData

Retrieves an application-specific data block that was associated with a player ID by using IDirectPlay3::SetPlayerData.

HRESULT GetPlayerData(
DPID *idPlayer*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*,
DWORD *dwFlags*
);

Parameters

idPlayer
ID of the player for which data is being requested.

lpData
Pointer to a buffer where the application-specific player data is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data.

lpdwDataSize
Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the group data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size required.

dwFlags
If this parameter is set to 0, the remote data will be retrieved.

DPGET_LOCAL
Retrieves the local data set by this application.

DPGET_REMOTE
Retrieves the current data from the remote shared data space.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDFLAGS
DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER

Remarks

DirectPlay can maintain two types of player data: local and remote. The application must specify which type of data to retrieve. Local data was set by this DirectPlay object by using the DPSET_LOCAL flag. Remote data might have been set by any application in the session by using the DPSET_REMOTE flag.

See Also

IDirectPlay3::SetPlayerData

IDirectPlay3::GetPlayerFlags

Returns the flags describing the player.

HRESULT GetPlayerFlags(
DPID *idPlayer*,
LPDWORD *lpdwFlags*
);

Parameters

idPlayer

The DPID of the player whose flag settings are to be retrieved.

lpdwFlags

Pointer to a DWORD to be set to the player's flag settings. Can be one or more of the following:

DPPLAYER_LOCAL – the player was created by this application. If this flag is not specified, the player is a remote player.

DPPLAYER_SERVERPLAYER – the player is a server player for client/server communications.

DPPLAYER_SPECTATOR – the player was created as a spectator.

Return Values

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

DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

See Also

IDirectPlay3::CreatePlayer

IDirectPlay3::GetPlayerName

Retrieves the name associated with a player.

HRESULT GetPlayerName(
DPID *idPlayer*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

idPlayer
ID of the player whose name is requested.

lpData
Pointer to a buffer where the name data is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data.

lpdwDataSize
Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the name data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size required.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER

Remarks

After this method returns, the pointer *lpData* should be cast to the DPNAME structure to read the group name data.

See Also

DPNAME, IDirectPlay3::SetPlayerName

IDirectPlay3::GetSessionDesc

Retrieves the properties of the current open session.

HRESULT GetSessionDesc(
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

lpData

Pointer to a buffer where the session description data is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data.

lpdwDataSize

Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the group data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size required. If this parameter is NULL, the method returns DPERR_INVALIDPARAM.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_NOCONNECTION

Remarks

After this method returns, the pointer *lpData* should be cast to the DPSESSIONDESC2 structure to read the session description data.

See Also

DPSESSIONDESC2, IDirectPlay3::EnumSessions, IDirectPlay3::Open

IDirectPlay3::Initialize

This method is provided for compliance with the COM protocol.

HRESULT Initialize(
    LPGUID *lpGUID*
    );

Parameters

lpGUID
Pointer to the globally unique identifier (GUID) used as the interface identifier.

Return Values

Returns DPERR_ALREADYINITIALIZED.

Remarks

Because the DirectPlay object is initialized when it is created, this method always returns the DPERR_ALREADYINITIALIZED return value.

See Also

IUnknown::AddRef, IUnknown::QueryInterface

IDirectPlay3::InitializeConnection

Initializes a DirectPlay connection. All the information about the connection, including the service provider to use, the network address of the server, and the GUID of the session, is passed in through the *lpConnection* parameter.

HRESULT InitializeConnection(
LPVOID *lpConnection*,
DWORD *dwFlags*
);

Parameters

lpConnection

Pointer to a buffer that contains all the information about the connection to be initialized as a DirectPlay Address.

dwFlags

Not used. Must be zero.

Return Values

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

DPERR_ALREADYINITIALIZED
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_UNAVAILABLE

This method returns DPERR_ALREADYINITIALIZED if **InitializeConnection** has previously been called on this object. It returns DPERR_UNAVAILABLE if the service provider could not be initialized. This can be because the resources necessary to operate this service provider are not present (for example, TCP/IP stack not present).

Remarks

The *lpConnection* parameter supplied to this method can be obtained from the IDirectPlay3::EnumConnections method, or the application can create one directly using the IDirectPlayLobby2::CreateCompoundAddress method.

This method loads and initializes the appropriate service provider. The *lpConnection* parameter is passed on to the service provider, which extracts and saves any relevant information. This information is used when the IDirectPlay3::EnumSessions or IDirectPlay3::Open method is called so that the service provider does not pop up a dialog box asking for that information.

This method replaces DirectPlayCreate as the means of binding a service provider to a DirectPlay object. The primary benefits of **InitializeConnection** are that you can override the service provider dialogs, and you can initialize a lobby provider.

See Also

IDirectPlay3::EnumConnections, IDirectPlayLobby2::CreateCompoundAddress, DirectPlay Address

IDirectPlay3::Open

Used to join a session that has been enumerated by a previous call to IDirectPlay3::EnumSessions or to create a new session that other users can enumerate and join.

HRESULT Open(
LPDPSESSIONDESC2 *lpsd*,
DWORD *dwFlags*
);

Parameters

lpsd

Pointer to the DPSESSIONDESC2 structure describing the session to be created or joined. If a session is being joined, then only the *dwSize*, *guidInstance*, and *lpszPassword* data members need to be specified. A password need only be supplied if the enumerated session had the DPSESSION_PASSWORD flag set.

If a session is being created, then the application must completely fill out the DPSESSIONDESC2 structure with the properties of the sessions to be created. The *guidInstance* will be generated by DirectPlay.

dwFlags

One and only one of the following flags:

DPOPEN_CREATE

Create a new instance of an application session. The local computer will be the name server and host of the session.

DPOPEN_JOIN

Join an existing instance of an application session for the purpose of participating. The application will be able to create players and send and receive messages.

DPOPEN_RETURNSTATUS

If this flag is specified, the method will not display any dialog boxes showing the connection progress status. If the connection cannot be made immediately, the method will return with the DPERR_CONNECTING error. The application must keep calling **Open** until either DP_OK is returned, indicating successful completion, or some other error code is returned, indicating an error.

Return Values

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

DPERR_ACCESSDENIED
DPERR_ALREADYINITIALIZED
DPERR_AUTHENTICATIONFAILED
DPERR_CANTLOADCAPI
DPERR_CANTLOADSECURITYPACKAGE
DPERR_CANTLOADSSPI
DPERR_CONNECTING
DPERR_ENCRYPTIONFAILED
DPERR_INVALIDCREDENTIALS
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_INVALIDPASSWORD
DPERR_LOGONDENIED
DPERR_NOCONNECTION
DPERR_NONEWPLAYERS
DPERR_SIGNFAILED

DPERR_TIMEOUT
DPERR_UNINITIALIZED
DPERR_USERCANCEL

This method returns DPERR_ALREADYINITIALIZED if there is already an open session on this DirectPlay object. It returns DPERR_TIMEOUT if the session did not respond to the **Open** request. It returns DPERR_USERCANCEL if the user canceled the enumeration process(usually by canceling a service provider dialog box).

Remarks

Once an application has joined a session, it can create a player and start communicating with other players in the session. The application will not receive any messages nor can it send any messages on this DirectPlay object until it creates a local player using IDirectPlay3::CreatePlayer.

In order to have two sessions open simultaneously, the application must create two DirectPlay objects and open a session on each one.

To join a session, it is only necessary to fill in the *dwSize* and *guidInstance* members of the DPSESSIONDESC2 structure. The *lpszPassword* member must also be filled in if the session was marked as password protected. An enumerated session will have the DPSESSION_PASSWORDREQUIRED flag set if it requires a password.

If joining a secure session, you must use SecureOpen to provide login credentials. The enumerated session will have the DPSESSION_SECURESERVER flag set if it requires credentials.

If you specify the DPSESSION_SECURESERVER flag in the DPSESSIONDESC2 structure, the session will be opened with the default security package, NTLM (NT LAN Manager). To specify an alternate security package, use SecureOpen.

When an application attempts to join a session, the server can reject the **Open** request or ignore it (in which case **Open** will time-out). Attempting to join a session where new players are disabled, joining is disabled, the player limit has been reached, or an incorrect password is supplied will result in a DPERR_NONEWPLAYERS or DPERR_INVALIDPASSWORD error.

See Also

DPSESSIONDESC2, IDirectPlay3::Close, IDirectPlay3::SecureOpen, IDirectPlay3::EnumSessions

IDirectPlay3::Receive

Retrieves a message from the message queue.

HRESULT Receive(
LPDPID *lpidFrom*,
LPDPID *lpidTo*,
DWORD *dwFlags*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

lpidFrom

Pointer to a DPID that will be set to the sender's player ID when this method returns. If the DPRECEIVE_FROMPLAYER flag is specified, this variable must be initialized with the player ID before calling this method.

lpidTo

Pointer to a DPID that will be set to the receiver's player ID when this method returns. If the DPRECEIVE_TOPLAYER flag is specified, this variable must be initialized with the player ID before calling this method.

dwFlags

One or more of the following control flags can be set. By default (*dwFlags* = 0), the first available message will be retrieved.

DPRECEIVE_ALL

Returns the first available message. This is the default.

DPRECEIVE_PEEK

Returns a message as specified by the other flags, but does not remove it from the message queue. This flag must be specified if *lpData* is NULL.

DPRECEIVE_TOPLAYER and **DPRECEIVE_FROMPLAYER**

If both DPRECEIVE_TOPLAYER and DPRECEIVE_FROMPLAYER are specified, **Receive** will only return messages that are 1) sent to the player specified by *lpidTo* and 2) sent from the player specified by *lpidFrom*. Note that both conditions must be met.

If only DPRECEIVE_TOPLAYER is specified, **Receive** will only return messages sent to the player specified by *lpidTo*.

If only DPRECEIVE_FROMPLAYER is specified, **Receive** will only return messages sent from the player specified by *lpidFrom*.

If neither DPRECEIVE_TOPLAYER nor DPRECEIVE_FROMPLAYER is set, **Receive** will return the first available message.

lpData

Pointer to a buffer where the message data is to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the size required to hold the data. If the message came from player ID DPID_SYSMSG, the application should cast *lpData* to DPMSG_GENERIC and check the **dwType** member to see what type of system message it is before processing it.

lpdwDataSize

Pointer to a DWORD that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the number of bytes copied into the buffer. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the buffer size required. The message order in the receive queue can change between calls to **IDirectPlay3::Receive**. Therefore, it is possible to get a DPERR_BUFFERTOOSMALL error again even after the application has allocated the memory requested from the previous call to **IDirectPlay3::Receive**. It is best to keep reallocating memory until a DPERR_BUFFERTOOSMALL error is not received.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_GENERIC
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER
DPERR_NOMESSAGES

Remarks

Any message received from a player ID defined as DPID_SYSMSG is a system message used to notify the application of a change in the session. In those cases, the *lpData* of system messages should be cast to DPMSG_GENERIC and the **dwType** member should be examined to see what specific system message it is.

Messages that were sent to a player ID defined as DPID_ALLPLAYERS or to a Group ID still appear to come from the sending player ID. An application will only receive messages directed to a local player. A player cannot receive a message in which the values pointed to by *lpidFrom* and *lpidTo* are equal.

If DPSESSION_NOMESSAGEID is specified in the session description, the *lpidFrom* will always be 0xFFFFFFFF and the *lpidTo* value is arbitrary.

All the service providers shipped with DirectPlay perform integrity checks on the data to protect against corruption. Any message retrieved using **Receive** is guaranteed to be free from corruption.

See Also

DPMSG_GENERIC, IDirectPlay3::Send

IDirectPlay3::SecureOpen

Creates or joins a secure session. When joining a secure session, use this method to supply login credentials.

When creating a new session, the host computer can specify an alternate security package to use. When joining a session, a computer can specify a user name and password.

HRESULT IDirectPlay3::SecureOpen(
LPCDPSESSIONDESC2 *lpsd*,
DWORD *dwFlags*,
LPCDPSECURITYDESC *lpSecurity*,
LPCDPCREDENTIALS *lpCredentials*
)

Parameters

lpsd

Pointer to the DPSESSIONDESC2 structure describing the session to be created or joined. If a session is to be joined, then only the guidInstance and **lpszPassword** members need to be specified. The password need only be supplied if the enumerated session had the DPSESSION_PASSWORDREQUIRED flag set.

If a session is to be created, then the application must completely fill out the DPSESSIONDESC2 structure with the properties of the sessions to be created. The *guidInstance* will be generated by DirectPlay. NOTE: The DPSESSION_SECURESERVER flag must be set to indicate that all computers attempting to open the session must be authenticated.

If you don't specify the DPSESSION_SECURESERVER flag when filling out the DPSESSIONDESC2 structure, **SecureOpen** will open an unsecure session, just as if you had called Open.

dwFlags

One of the following flags:

DPOPEN_CREATE

Creates a new instance of a secured session.

DPOPEN_JOIN

Joins an existing instance of a secured session.

DPOPEN_RETURNSTATUS

If this flag is specified, the method will not display any dialog boxes showing the connection progress status. If the connection cannot be made immediately, the method will return with the DPERR_CONNECTING error. The application must keep calling **SecureOpen** until either DP_OK is returned, indicating successful completion, or some other error code is returned, indicating an error.

lpSecurity

Pointer to a DPSECURITYDESC structure containing the security package to use. Set this parameter to NULL to use the default security package (NT LAN Manager) and CryptoAPI package (Microsoft RSA Base Cryptographic Provider). Relevant only when creating a session. Must be set to NULL when joining a session.

lpCredentials

Pointer to a DPCREDENTIALS structure containing the logon name, password, and domain to be authenticated on the server. NULL if there are no credentials. Credentials are ignored when creating a session.

Return Values

Returns DP_OK if successful or one of the following error values:

DPERR_ACCESSDENIED
DPERR_ALREADYINITIALIZED

DPERR_AUTHENTICATIONFAILED
DPERR_CANTLOADCAPI
DPERR_CANTLOADSECURITYPACKAGE
DPERR_CANTLOADSSPI
DPERR_CONNECTING
DPERR_ENCRYPTIONFAILED
DPERR_INVALIDCREDENTIALS
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_INVALIDPASSWORD
DPERR_LOGONDENIED
DPERR_NOCONNECTION
DPERR_NONEWPLAYERS
DPERR_SIGNFAILED
DPERR_TIMEOUT
DPERR_UNINITIALIZED
DPERR_USERCANCEL

This method DPERR_LOGONDENIED after being called with invalid creadentials or without credentials when credentials are required. The application must collect the user's credentials and call **SecureOpen** again.

Remarks

When joining a session, first call **SecureOpen** with no credentials. If the method returns DPERR_LOGONDENIED, then the application must collect the user name and password from the user and call **SecureOpen** again, passing in the user's credentials through the *lpCredentials* parameter. If the method returns DP_OK, then the player was able to login with the credentials he or she had specified earlier during system logon (network logon in NTLM).

See Also

IDirectPlay3::Open

IDirectPlay3::Send

Sends a message to another player, to a group of players, or to all players in the session. To send a message to another player, specify the target player's player ID. To send a message to a group of players, send the message to the group ID assigned to the group. To send a message to the entire session, send the message to the player ID DPID_ALLPLAYERS. Messages can be sent using either a guaranteed or nonguaranteed protocol on a per message basis. If the session is being hosted on a secure server, messages can be sent encrypted (to ensure privacy) or digitally signed (to ensure authenticity) on a per message basis.

HRESULT Send(
DPID *idFrom*,
DPID *idTo*,
DWORD *dwFlags*,
LPVOID *lpData*,
DWORD *dwDataSize*
);

Parameters

idFrom

ID of the sending player. The player ID must correspond to one of the local players on this computer.

idTo

The destination ID of the message. To send a message to another player, specify the ID of the player. To send a message to all the players in a group, specify the ID of the group. To send a message to all the players in the session, use the constant symbol DPID_ALLPLAYERS. To send a message to the server player, specify the constant symbol DPID_SERVERPLAYER. A player cannot send a message to itself.

dwFlags

Indicates how the message should be sent. By default (*dwFlags* = 0), the message is sent nonguaranteed.

DPSEND_ENCRYPTED

Sends the messages encrypted. This can only be done in a secure session. This flag can only be used if the **DPSEND_GUARANTEED** flag is also set. The message will be sent as a DPMSG_SECUREMESSAGE system message.

DPSEND_GUARANTEED

Sends the message by using a guaranteed method of delivery if it is available.

DPSEND_SIGNED

Sends the message with a digital signature. This can only be done in a secure session. This flag can only be used if the **DPSEND_GUARANTEED** flag is also set. The message will be sent as a DPMSG_SECUREMESSAGE system message.

lpData

Pointer to the data being sent.

dwDataSize

Length of the data being sent.

Return Values

Returns DP_OK if successful or one of the following error values:

DPERR_BUSY
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER
DPERR_NOTLOGGEDIN
DPERR_SENDTOOBIG

This method returns DPERR_INVALIDPARAMS if the encrypted or signed flag is specified for a message that is not also specified as guaranteed. It returns DPERR_NOTLOGGEDIN when the client application tries to send a secure message without first logging in.

Remarks

Messages can be sent guaranteed or nonguaranteed. By default, messages are sent nonguaranteed which means that DirectPlay does no verification that the message reached the intended recipient. Sending a guaranteed message takes much longer; a minimum of 2 to 3 times longer than nonguaranteed messages. Applications should try to minimize sending guaranteed messages as much as possible and design the application to tolerate lost messages. All the service providers shipped with DirectPlay perform integrity checks on the data to protect against corruption. Any message retrieved using this method is guaranteed to be free from corruption.

A player cannot send a message to itself. If a player sends a message to a group that it is part of or to DPID_ALLPLAYERS, it will not receive a copy of that message. The exception to this rule is if the DPSESSION_NOMESSAGEID was specified in the session description (DPSESSIONDESC2). Then it is possible for a player to receive a message that it sent to a group. Because there is no DirectPlay message ID header on the message (indicating who sent the message), it cannot filter out messages based on the message ID.

When DPSESSION_NOMESSAGEID is used, the *idFrom* parameter has no meaning and the *idTo* parameter is used simply to direct the message to the correct target computer. If the target computer has more than one player on it, it cannot be determined whose receive queue the message will appear in. When the message is received, it will appear to have come from player DPID_UNKNOWN.

There is no limit to the size of messages that can be transmitted using the **Send** method. DirectPlay will automatically break up large messages into packets (packetize) and reassemble them on the receiving end. Beware of sending large messages nonguaranteed — if even one of the packets fails to reach the receiver then the entire message will be ignored. The application can determine the maximum size of a message before it starts packetizing by calling GetCaps and examining the dwMaxBufferSize member of the DPCAPS structure.

When you send an encryted or signed message, it is not delivered as an application message, but as a system message, DPMSG_SECUREMESSAGE.

See Also

IDirectPlay3::Receive, IDirectPlay3::SendChatMessage, DPMSG_SECUREMESSAGE

IDirectPlay3::SendChatMessage

Sends a text message to another player, a group of players, or all players. This method supports both Unicode (the IDirectPlay3 interface) and ANSI strings (the IDirectPlay3A interface). The player receiving the chat message is informed through a DPMSG_CHAT system message in the player's receive queue. This method must be used in a lobby session.

HRESULT SendChatMessage(
DPID *idFrom*,
DPID *idTo*,
DWORD *dwFlags*,
LPDPCHAT *lpChatMessage*
);

Parameters

idFrom
ID of the sending player. The player ID must correspond to one of the local players on this computer.

idTo
ID of the player to send the message to, the group ID of the group of players to send the message to, or DPID_ALLPLAYERS to send the message to all players in the session.

dwFlags
Indicates how the message should be sent. If this parameter is set to 0, the message is sent nonguaranteed.

DPSEND_GUARANTEED
Sends the message by using a guaranteed method of delivery if it is available.

lpChatMessage
Pointer to a DPCHAT structure containing the message to be sent.

Return Values

Returns DP_OK if successful or one of the following error values:

DPERR_ACCESSDENIED
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

This method returns DPERR_INVALIDPARAMS if the *idTo* ID is not a valid player or group. It returns DPERR_INVALIDPLAYER if the *idFrom* ID is not a valid player. It returns DPERR_ACCESSDENIED if the *idFrom* ID is not a local player.

Remarks

This method facilitates player to player chatting within a lobby session where it is possible for different client applications to be connected. You must use this method in a lobby session. Use is optional in an application seession.

The receiving player will receive a system message (*idFrom* = DPID_SYSTEM). The DPCHAT structure will specify which player the chat message came from.

See Also

DPCHAT, DPMSG_CHAT, IDirectPlay3::Send

IDirectPlay3::SetGroupConnectionSettings

Sets the connection settings for a session that will be launched from this group. This method can only be used in a lobby session.

HRESULT SetGroupConnectionSettings(
DWORD *dwFlags*,
DPID *idGroup*,
LPDPLCONNECTION *lpConnection*
);

Parameters

dwFlags
Not used. Must be zero.

idGroup
The DPID of the group to set the connection settings on.

lpConnection
Pointer to a DPLCONNECTION structure describing the application to be launched, the service provider to use, and the session description of the session to be created.

Return Values

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

DPERR_ACCESSDENIED
DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS
DPERR_UNSUPPORTED

Remarks

Call IDirectPlay3::GetGroupConnectionSettings before calling **SetGroupConnectionSettings** to see if any of the DPLCONNECTION structure members already have default values (non-NULL or non-zero). If so, you may get an error if you try to change these default values.

You do not have to set the *lpAddress* and *dwAddressSize* data members of the DPLCONNECTION structure with **SetGroupConnectionSettings**. In the DPSESSIONDESC2 structure within the DPLCONNECTION structure, you do not have to fill in the *guidInstance* member.

See Also

DPLCONNECTION, IDirectPlayLobby2::RunApplication, IDirectPlay3::GetGroupConnectionSettings

IDirectPlay3::SetGroupData

Associates an application-specific data block with a group ID. Only the computer that created the group can change the remote data associated with it.

HRESULT SetGroupData(
DPID *idGroup*,
LPVOID *lpData*,
DWORD *dwDataSize*,
DWORD *dwFlags*
);

Parameters

idGroup
Group ID for which data is being set.

lpData
Pointer to the data to be set. Set to NULL to clear any existing group data.

dwDataSize
Size of the data buffer. If *lpData* is NULL and this parameter does not equal zero, the method returns DPERR_INVALIDPARAMS.

dwFlags
If this parameter is set to 0, the remote group data will be set and propagated using nonguaranteed messaging.

DPSET_GUARANTEED
Propagates the data by using guaranteed messaging (if available). This flag can only be used with DPSET_REMOTE.

DPSET_LOCAL
This data is for local use only and will not be propagated.

DPSET_REMOTE
This data is for use by all the applications, and will be propagated to all the other applications in the session. This flag can only be used on groups owned by the local session.

Return Values

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

DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

DirectPlay can maintain two types of group data: local and remote. Local data is available only to the application on the local computer. Remote data is propagated to all the other applications in the session. A DPSYS_SETPLAYERORGROUPDATA system message will be sent to all the other players notifying them of the change unless DPSESSION_NODATAMESSAGES is set in the session description. It is safe to store pointers to resources in the local data; the local data block is available (in the DPMSG_DESTROYPLAYERORGROUP system message) when the group is being destroyed, so the application can free those resources. For a list of system messages, see Using System Messages.

This method should not be used for updating real-time information (such as position updates) due to the overhead introduced. It is much more efficient to use IDirectPlay3::Send for this. **IDirectPlay3::SetGroupData** is more appropriate for shared state information that doesn't change very often and is not time critical (such as a team color).

See Also

DPMSG_SETPLAYERORGROUPDATA, IDirectPlay3::GetGroupData, IDirectPlay3::Send

IDirectPlay3::SetGroupName

Sets the name of a group after it has been created. Only the computer that created the group can set the name of the group. A DPMSG_SETPLAYERORGROUPNAME system message will be sent to all the other players notifying them of the change unless DPSESSION_NODATAMESSAGES is set in the session description.

HRESULT SetGroupName(
DPID *idGroup*,
LPDPNAME *lpGroupName*,
DWORD *dwFlags*
);

Parameters

idGroup
ID of the group for which the name is being set.

lpGroupName
Pointer to a DPNAME structure containing the name information for the group. Set this parameter to NULL if the group has no name information.

dwFlags
If this parameter is set to 0, the name will be propagated to all the remote systems by using nonguaranteed message passing. This value can only be used on groups owned by the local session.

DPSET_GUARANTEED
Propagates the data using guaranteed messaging (if available).

DPSET_LOCAL
This data is for local use only and will not be propagated.

DPSET_REMOTE
This data is for use by all the applications, and will be propagated to all the other applications in the session. This flag can only be used on groups owned by the local session.

Return Values

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

DPERR_INVALIDGROUP
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

See Also

DPNAME, DPMSG_SETPLAYERORGROUPNAME, IDirectPlay3::GetGroupName, IDirectPlay3::Send

IDirectPlay3::SetPlayerData

Associates an application-specific data block with a player ID.

HRESULT SetPlayerData(
DPID *idPlayer*,
LPVOID *lpData*,
DWORD *dwDataSize*,
DWORD *dwFlags*
);

Parameters

idPlayer
ID of the player for which data is being set.

lpData
Pointer to the data to be set. Set this parameter to NULL and *dwDataSize* to zero to clear out any existing player data.

dwDataSize
Size of the data buffer. If *lpData* is NULL and this parameter does not equal zero, the method returns DPERR_INVALIDPARAMS.

dwFlags
If this parameter is set to 0, the remote player data will be set and propagated by using nonguaranteed messaging.

DPSET_GUARANTEED
Propagates the data by using guaranteed messaging (if available). This flag can only be used with DPSET_REMOTE.

DPSET_LOCAL
This data is for local use only and will not be propagated.

DPSET_REMOTE
This data is for use by all the applications, and will be propagated to all the other applications in the session.

Return Values

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

DPERR_ACCESSDENIED
DPERR_INVALIDFLAGS
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_INVALIDPLAYER

Remarks

SetPlayerData should not be used for updating real-time information (such as position updates) due to the overhead introduced. It is much more efficient to use Send for this. **SetPlayerData** is more appropriate for shared state information that doesn't change very often and is not time critical (such as a player's name).

DirectPlay can maintain two types of player data: local and remote. Local data is available only to the application on the local computer. Remote data is propagated to all the other applications in the session. This method returns DPERR_ACCESSDENIED if you try to set player data for a remote player. A DPMSG_SETPLAYERORGROUPDATA system message will be sent to all the other players notifying them of the change unless DPSESSION_NODATAMESSAGES is set in the session description. It is safe to store pointers to resources in the local data; the local data block is available (in the DPMSG_DESTROYPLAYERORGROUP system message) when the player is being destroyed, so the application can free those resources. For a list of system messages, see Using System Messages.

See Also

DPMSG_SETPLAYERORGROUPDATA, IDirectPlay3::GetPlayerData, IDirectPlay3::Send

IDirectPlay3::SetPlayerName

Sets the name of a local player after it has been changed. Only the computer that created the player can change the name. A DPMSG_SETPLAYERORGROUPNAME system message will be sent to all the other players notifying them of the change unless DPSESSION_NODATAMESSAGES is set in the session description.

HRESULT SetPlayerName(
DPID *idPlayer*,
LPDPNAME *lpPlayerName*,
DWORD *dwFlags*
);

Parameters

idPlayer
ID of the local player for which data is being sent.

lpPlayerName
Pointer to a DPNAME structure containing the name information for the player. Set this parameter to NULL if the player has no name information.

dwFlags
Flags indicating how the name will be propagated. It can be one of the following values:

DPSET_GUARANTEED
Propagates the data by using guaranteed messaging (if available).

DPSET_LOCAL
Data is not propagated to other players.

DPSET_REMOTE
Propagates the data to all players in the session using nonguaranteed message passing. This is the default value.

Return Values

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

DPERR_INVALIDOBJECT
DPERR_INVALIDPLAYER

See Also

DPNAME, DPMSG_SETPLAYERORGROUPNAME, IDirectPlay3::GetPlayerName, IDirectPlay3::Send

IDirectPlay3::SetSessionDesc

Changes the properties of the current session. Only the host of the session can change the session properties. If a nonhost attempts to call it, the method returns DPERR_ACCESSDENIED.

The updated session description will be propagated to all the other computers in the session. Each player will receive a DPMSG_SETSESSIONDESC system message.

You can't use **SetSessionDesc** in a lobby session.

HRESULT SetSessionDesc(
LPDPSESSIONDESC2 *lpSessDesc*,
DWORD *dwFlags*
);

Parameters

lpSessDesc
Pointer to the DPSESSIONDESC2 structure containing the new settings.

dwFlags
No flags are currently used by this method.

Return Values

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

DPERR_ACCESSDENIED
DPERR_INVALIDPARAMS
DPERR_NOSESSIONS

This method returns DPERR_ACCESSDENIED if this computer does not have permission to change the session. Only the host can change the session properties. It returns DPERR_INVALIDPARAMS if the application attempts to change a property in the session description that cannot be changed or tries to give a property an invalid value.

Remarks

Changing the session description will cause a DPSYS_SETSESSIONDESC system message to be generated on all the other computers in the session.

The following members and flags of the DPSESSIONDESC2 structure can be changed using **IDirectPlay3::SetSessionDesc**:

dwFlags
DPSESSION_JOINDISABLED
DPSESSION_MIGRATEHOST
DPSESSION_NEWPLAYERSDISABLED
DPSESSION_NODATAMESSAGES
DPSESSION_PRIVATE

dwMaxPlayers
If you set the maximum players to a value less than the current number of players, the method returns DPERR_INVALIDPARAMS.

lpszSessionName / *lpszSessionNameA*

lpszPassword / *lpszPasswordA*

dwUser1

dwUser2

dwUser3

dwUser4

If the following members and flags of the DPSESSIONDESC2 structure are changed, an error will

occur (for example, if the DPSESSION_KEEPALIVE flag is currently set and you try to set this bit, you will not get an error, but if you try to clear this bit, you will get an error and **IDirectPlay3::SetSessionDesc** will fail):

dwSize

dwFlags

DPSESSION_NOMESSAGEID
DPSESSION_KEEPALIVE

The following members of the DPSESSIONDESC2 structure are ignored (that is, it does not matter what you pass in these members — DirectPlay will always use the values passed when IDirectPlay3::Open was called):

guidInstance

guidApplication

dwCurrentPlayers

dwReserved1

dwReserved2

See Also

DPSESSIONDESC2, IDirectPlay3::GetSessionDesc

IDirectPlay3::StartSession

Use this method to initiate the launch of a DirectPlay session. Call SetGroupConnectionSettings first to specify what application to launch, which service provider to use, and what session description to use.

When **StartSession** is called, each player in the group receives a DPMSG_STARTSESSION instructing the player to launch the session.

HRESULT StartSession(
DWORD *dwFlags*,
DPID *idGroup*
);

Parameters

dwFlags
Not used. Must be zero.

idGroup
The DPID of the group to send start session commands to. The group must be a staging area.

Return Values

Returns DP_OK if successful or one of the following error values:

DPERR_ACCESSDENIED
DPERR_INVALIDFLAGS
DPERR_INVALIDGROUP
DPERR_INVALIDPARAMS

This method returns DPERR_INVALIDGROUP if the given group ID is not a valid staging area.

Remarks

A player joining a staging area group for which the session has already started can call this method to join the session. The player will receive a DPMSG_STARTSESSION message instructing it how to join the session.

You can determine if a session is already in progress by calling IDirectPlay3::GetGroupConnectionSettings and examining the *guidInstance* parameter it returns through a pointer in the DPLCONNECTION structure to the DPSESSIONDESC2 structure. If *guidInstance* is GUID_NULL, no session is in progress. If it is any other value, a session is already in progress.

See Also

DPMSG_STARTSESSION

IDirectPlayLobby2

Applications use the methods of the **IDirectPlayLobby2** interface to manage applications and their associated data. This section is a reference to the methods of this interface. For a conceptual overview, see IDirectPlayLobby2 Interface.

Address management	CreateAddress CreateCompoundAddress EnumAddress EnumAddressTypes
Application management	Connect EnumLocalApplications RunApplication
Data management	GetConnectionSettings ReceiveLobbyMessage SendLobbyMessage SetConnectionSettings SetLobbyMessageEvent

IDirectPlayLobby2::Connect

Connects an application to the session specified by the DPLCONNECTION structure currently stored with the DirectPlayLobby object.

Note This method will return an **IDirectPlay2** or **IDirectPlay2A** interface. Use the standard COM **QueryInterface** method to obtain an IDirectPlay3 or IDirectPlay3A method.

HRESULT Connect(
DWORD *dwFlags*,
LPDIRECTPLAY2 **lplpDP*,
IUnknown FAR **pUnk*
);

Parameters

dwFlags
Reserved; must be zero.

lplpDP
Pointer to a pointer to be initialized with a valid interface — either **IDirectPlay2** (if called on IDirectPlayLobby2) or **IDirectPlay2A** (if called on **IDirectPlayLobby2A**).

pUnk
Pointer to the containing *IUnknown* interface. This parameter is provided for future compatibility with COM aggregation features. Presently, however, **IDirectPlayLobby2::Connect** returns an error if this parameter is anything but NULL.

Return Values

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

CLASS_E_NOAGGREGATION
DPERR_INVALIDFLAGS
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_NOTLOBBIED
DPERR_OUTOFMEMORY

Remarks

After this method is successfully completed, the application can skip the process of calling IDirectPlay3::InitializeConnection, IDirectPlay3::EnumSessions, and IDirectPlay3::Open. The application should not ask the user a name but instead create a player using the player name information in the DPLCONNECTION structure.

Before calling this method, the application can examine the connection settings that will be used to start the application by using the IDirectPlayLobby2::GetConnectionSettings method. The application then can modify these settings and set them by using the IDirectPlayLobby2::SetConnectionSettings method. The application should pay particular attention to the DPSESSIONDESC2 structure to ensure that the proper session properties are set, especially **dwFlags**, **dwMaxPlayers**, and the **dwUser** members.

IDirectPlayLobby2::CreateAddress

Creates a DirectPlay Address, given a service provider-specific network address. The resulting address contains the globally unique identifier (GUID) of the service provider and data that the service provider can interpret as a network address.

For more information about the DirectPlay Address, see DirectPlay Address. For a list of predefined Microsoft data types, see DirectPlay Address Data Types.

HRESULT CreateAddress(
REFGUID *guidSP*,
REFGUID *guidDataType*,
LPCVOID *lpData*,
DWORD *dwDataSize*,
LPVOID *lpAddress*,
LPDWORD *lpdwAddressSize*
);

Parameters

guidSP
Pointer to the GUID of the service provider. (In C++, it is a reference to the GUID.)

guidDataType
Pointer to the GUID identifying the specific network address type being used. (In C++, it is a reference to the GUID.) For information about predefined network address types, see DirectPlay Address.

lpData
Pointer to a buffer containing the specific network address.

dwDataSize
Size, in bytes, of the network address in *lpData*.

lpAddress
Pointer to a buffer in which the constructed DirectPlay Address is to be written.

lpdwAddressSize
Pointer to a variable containing the size of the DirectPlay Address buffer. Before calling this method, the service provider must initialize *lpdwAddressSize* to the size of the buffer. After the method has returned, this parameter will contain the number of bytes written to *lpAddress*. If the buffer was too small (DPERR_BUFFERTOOSMALL), this parameter will be set to the size required to store the DirectPlay Address.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDPARAMS

Remarks

The IDirectPlayLobby2::CreateCompoundAddress method can be used to create longer DirectPlay Addresses than **CreateAddress** allows.

See Also

IDirectPlayLobby2::EnumAddress, IDirectPlayLobby2::CreateCompoundAddress

IDirectPlayLobby2::CreateCompoundAddress

Creates a DirectPlay Address from a list of individual data chunks. This method can be used to create longer DirectPlay Addresses than IDirectPlayLobby2::CreateAddress allows. For a list of predefined Microsoft data types, see DirectPlay Address Data Types.

HRESULT CreateCompoundAddress(
LPDPCOMPOUNDADDRESSELEMENT *lpElements*,
DWORD *dwElementCount*,
LPVOID *lpAddress*,
LPDWORD *lpdwAddressSize*
);

Parameters

lpElements
Pointer to the first element in an array of DPCOMPOUNDADDRESSELEMENT structures that will be used to generate the DirectPlay Address.

dwElementCount
The number of address elements in the array pointed to by the *lpElements* parameter.

lpData
Pointer to a buffer that the complete DirectPlay Address is to be written to. Pass NULL if only the required size of the buffer is desired.

lpdwDataSize
Pointer to a DWORD with the size of the *lpData* buffer. The DWORD will be modified to reflect the actual number of bytes copied into the buffer. If the buffer was too small, it will contain the number of bytes required.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_INVALIDFLAGS
DPERR_INVALIDPARAMS

See Also

IDirectPlayLobby2::CreateAddress, IDirectPlayLobby2::EnumAddress, DPCOMPOUNDADDRESSELEMENT

IDirectPlayLobby2::EnumAddress

Parses out chunks from the DirectPlay Address buffer.

HRESULT EnumAddress(
LPDPENUMADDRESSCALLBACK *lpEnumAddressCallback*,
LPCVOID *lpAddress*,
DWORD *dwAddressSize*,
LPVOID *lpContext*
);

Parameters

lpEnumAddressCallback
Pointer to a EnumAddressCallback function that will be called for each information chunk in the DirectPlay Address.

lpAddress
Pointer to the start of the DirectPlay Address buffer.

dwAddressSize
Size of the DirectPlay Address.

lpContext
Context that will be passed to the callback function.

Return Values

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

DPERR_EXCEPTION
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS

Remarks

For more information about the DirectPlay Address, see DirectPlay Address.

See Also

IDirectPlayLobby2::CreateAddress

IDirectPlayLobby2::EnumAddressTypes

Enumerates all the address types that a given service provider needs to build the DirectPlay Address. The application or lobby can use this information to obtain the correct information from the user and create a DirectPlay Address.

HRESULT EnumAddressTypes(
LPDPLENUMADDRESSTYPESCALLBACK *lpEnumAddressTypeCallback***,**
REFGUID *guidSP***,**
LPVOID *lpContext***,**
DWORD *dwFlags*
);

Parameters

lpEnumAddressTypeCallback
Pointer to the EnumAddressTypeCallback function that will be called for each address type for a service provider. If the service provider takes no address type, the callback will not be called.

guidSP
Pointer to the GUID of the service provider whose address types are to be enumerated. (In C++, it is a reference to the GUID.)

lpContext
Context that will be passed to the callback function.

dwFlags
Reserved; must be zero.

Return Values

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

DPERR_EXCEPTION
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS

Remarks

For more information about the DirectPlay Address, DirectPlay Address. You can use **EnumAddressTypes** to determine if a service provider displays dialog boxes prompting the user for information; for example, a dialog box that asks for an IP address. If the service provider takes no address types, then it needs no information and will not display the dialog boxes.

An application can call IDirectPlay3::GetPlayerAddress to obtain a list of valid choices for an address type. This is only available for the modem-to-modem service providers. DirectPlay Address data types that are null-terminated Unicode strings end in W (for example, DPAID_INetW), while DirectPlay Address data types that are null-terminated ANSI strings do not (for example, DPAID_INet). For a list of predefined Microsoft data types, see DirectPlay Address Data Types.

See Also

DirectPlay Address, IDirectPlayLobby2::CreateAddress

IDirectPlayLobby2::EnumLocalApplications

Enumerates what applications are registered with DirectPlay.

HRESULT EnumLocalApplications(
LPDPENUMLOCALAPPLICATIONSCALLBACK *lpEnumLocalAppCallback*,
LPVOID *lpContext*,
DWORD *dwFlags*
);

Parameters

lpEnumLocalAppCallback
Pointer to the EnumLocalApplicationsCallback function that will be called for each enumerated application.

lpContext
Context passed to the callback function.

dwFlags
Reserved; must be zero.

Return Values

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

DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY

See Also

DPLAPPINFO

IDirectPlayLobby2::GetConnectionSettings

Retrieves the DPLCONNECTION structure that contains all the information needed to start and connect an application. The data returned is the same data that was passed to the IDirectPlayLobby2::RunApplication method by the lobby client, or set by calling the IDirectPlayLobby2::SetConnectionSettings method.

HRESULT GetConnectionSettings(
DWORD *dwAppID*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

dwAppID

Identifies which application's connection settings to retrieve when called from a lobby client (that communicates with several applications). When called from an application (that only communicates with one lobby client), this parameter must be zero. This ID number is obtained from IDirectPlayLobby2::RunApplication.

lpData

Pointer to a buffer in which the connection settings are to be written. Set this parameter to NULL to request only the size of data. The *lpdwDataSize* parameter will be set to the minimum size required to hold the data.

lpdwDataSize

Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the data. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the minimum buffer size required.

Return Values

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

DPERR_BUFFERTOOSMALL
DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_NOTLOBBIED
DPERR_OUTOFMEMORY

Remarks

The *lpData* member should be cast to the DPLCONNECTION structure when the function returns to read the data from it.

See Also

DPLCONNECTION, IDirectPlayLobby2::RunApplication, IDirectPlayLobby2::SetConnectionSettings

IDirectPlayLobby2::ReceiveLobbyMessage

Retrieves the message sent between a lobby client application and a DirectPlay application. Messages are queued, so there is no danger of losing data if it is not read in time.

HRESULT ReceiveLobbyMessage(
DWORD *dwFlags*,
DWORD *dwAppID*,
LPDWORD *lpdwMessageFlags*,
LPVOID *lpData*,
LPDWORD *lpdwDataSize*
);

Parameters

dwFlags

Reserved; must be zero.

dwAppID

Identifies which application's message to retrieve when called from a lobby client (that communicates with several applications). When called from an application (that communicates only with one lobby client), this parameter must be set to zero. This ID number is obtained by using the IDirectPlayLobby2::RunApplication method.

lpdwMessageFlags

Flags indicating what type of message is being returned. The default (*lpdwMessageFlags* = 0) indicates that the message is custom-defined by the sender. Processing of this type of message is optional. The receiver must interpret this message based on the identity of the sending application. A lobby can identify the sending application based on the GUID of the application that was launched. An application will need to identify the lobby by sending the lobby a standard message requesting an identifying GUID.

DPLMSG_STANDARD

Indicates that this is a DirectPlay-defined message. Processing of this type of message is optional.

DPLMSG_SYSTEM

Indicates that this is a DirectPlay generated system message used to inform the lobby of changes in the status of the application it launched.

lpData

Pointer to a buffer in which the message is to be written. Set this parameter to NULL to request only the size of message. The *lpdwDataSize* parameter will be set to the minimum size required to hold the message.

lpdwDataSize

Pointer to a variable that is initialized to the size of the buffer before calling this method. After the method returns, this parameter will be set to the size, in bytes, of the message. If the buffer was too small (DPERR_BUFFERTOOSMALL), then this parameter will be set to the minimum buffer size required.

Return Values

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

DPERR_APPNOTSTARTED
DPERR_BUFFERTOOSMALL
DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_NOMESSAGES

DPERR_OUTOFMEMORY

See Also

IDirectPlayLobby2::RunApplication, IDirectPlayLobby2::SendLobbyMessage

IDirectPlayLobby2::RunApplication

Starts an application and passes to it all the information necessary to connect it to a session. This method is used by a lobby client.

HRESULT RunApplication(
DWORD *dwFlags*,
LPDWORD *lpdwAppID*,
LPDPLCONNECTION *lpConn*,
HANDLE *hReceiveEvent*
);

Parameters

dwFlags
Reserved; must be zero.

lpdwAppId
Pointer to a variable that will be filled with an ID identifying the application that was started. The lobby client must save this application ID for use on with any calls to the IDirectPlayLobby2::SendLobbyMessage and IDirectPlayLobby2::ReceiveLobbyMessage methods.

lpConn
Pointer to a DPLCONNECTION structure that contains all the information necessary to specify which application to start and how to get it connected to a session instance without displaying any user dialog boxes.

hReceiveEvent
Specifies a synchronization event that will be set when a lobby message is received. This event can be changed later by using the IDirectPlayLobby2::SetLobbyMessageEvent method.

Return Values

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

DPERR_CANTCREATEPROCESS
DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY
DPERR_UNKNOWNAPPLICATION

Remarks

This method will return after the application process has been created. The lobby client will receive a system message indicating the status of the application. If the lobby client is starting an application that will be hosting a session, it should wait until it receives a DPLSYS_DPLAYCONNECTSUCCEEDED system message before starting the other applications that will be joining the session. If the application was unable to create or join a session, a DPLSYS_DPLAYCONNECTFAILED message will be generated. The lobby client will also receive a DPLSYS_CONNECTIONSETTINGSREAD system message when the application has read the connection settings and a DPLSYS_APPTERMINATED system message when the application terminates.

It is important that the lobby client not release its IDirectPlayLobby2 interface before it receives a DPLSYS_CONNECTIONSETTINGSREAD system message. The lobby client can either check IDirectPlayLobby2::ReceiveLobbyMessage in a loop until it is received, or supply a synchronization event.

See Also

IDirectPlayLobby2::ReceiveLobbyMessage, IDirectPlayLobby2::GetConnectionSettings, IDirectPlayLobby2::SetLobbyMessageEvent

IDirectPlayLobby2::SendLobbyMessage

Sends a message between the application and the lobby client.

HRESULT SendLobbyMessage(
DWORD *dwFlags*,
DWORD *dwAppID*,
LPVOID *lpData*,
DWORD *dwDataSize*
);

Parameters

dwFlags

Flags indicating the type of message being sent. The default (*dwFlags* = 0) is a custom message defined by the application sending it. Other possible values are:

DPLMSG_STANDARD – this is a standard message defined by DirectPlay.

dwAppID

Identifies which application to send a message to when called from a lobby client (that communicates with several applications). When called from an application (that communicates with only one lobby client), this parameter must be zero. This ID is obtained by using the IDirectPlayLobby2::RunApplication method.

lpData

Pointer to the buffer containing the message to send.

dwDataSize

Size, in bytes, of the buffer.

Return Values

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

DPERR_APPNOTSTARTED
DPERR_BUFFERTOOLARGE
DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY
DPERR_TIMEOUT

See Also

IDirectPlayLobby2::RunApplication, IDirectPlayLobby2::ReceiveLobbyMessage, DPLMSG_SETPROPERTY, DPLMSG_SETPROPERTYRESPONSE, DPLMSG_GETPROPERTY, DPLMSG_GETPROPERTYRESPONSE,

IDirectPlayLobby2::SetConnectionSettings

Modifies the DPLCONNECTION structure, which contains all the information needed to start and connect an application.

HRESULT SetConnectionSettings(
DWORD *dwFlags*,
DWORD *dwAppID*,
LPDPLCONNECTION *lpConn*
);

Parameters

dwFlags

Reserved; must be zero.

dwAppID

When called from a lobby client (that communicates with several applications), this parameter identifies which application's connection settings to retrieve. When called from an application (that communicates with only one lobby client), this parameter must be zero. This ID is obtained by using the IDirectPlayLobby2::RunApplication method.

lpConn

Pointer to a DPLCONNECTION structure that contains all the information necessary to specify which application to start and how to get it connected to a session instance without displaying any user dialog boxes.

Return Values

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

DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY

See Also

IDirectPlayLobby2::GetConnectionSettings

IDirectPlayLobby2::SetLobbyMessageEvent

Registers an event that will be set when a lobby message is received. The application must call this method if it needs to synchronize with messages. The lobby client can call this method to change the events specified in the call to the IDirectPlayLobby2::RunApplication method.

HRESULT SetLobbyMessageEvent(
DWORD *dwFlags*,
DWORD *dwAppID*,
HANDLE *hReceiveEvent*
);

Parameters

dwFlags
Reserved; must be zero.

dwAppID
Identifies which application the event is associated with when called from a lobby client (that communicates with several applications). When called from an application (that communicates with only one lobby client), this parameter must be zero. This ID number is obtained from IDirectPlayLobby2::RunApplication.

hReceiveEvent
Event handle to be set when a message is received.

Return Values

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

DPERR_GENERIC
DPERR_INVALIDINTERFACE
DPERR_INVALIDOBJECT
DPERR_INVALIDPARAMS
DPERR_OUTOFMEMORY

See Also

IDirectPlayLobby2::ReceiveLobbyMessage, IDirectPlayLobby2::SendLobbyMessage

Structures

Some structures have changed for DirectPlay 5, with new data members added to the end. Applications that use the **IDirectPlay2** interface but are compiled using the DirectX 5 header files can have problems. An application might crash if it is run on a computer that has the DirectX 3 run time installed and the application references data members that were added for DirectX 5.

Applications that need to be backward compatible with older run times should either:

- use the structures that existed in the DirectX 3 header files
- check the *dwSize* member in each structure before attempting to access to members of the structure

The structures that have new members for DirectPlay 5 are:

Structure	**New Members**
DPCREDENTIALS	union of LPWSTR lpszDomain LPSTR lpszDomainA

The DirectPlay structures are:

- DPACCOUNTDESC
- DPCAPS
- DPCHAT
- DPCOMPORTADDRESS
- DPCOMPOUNDADDRESSELEMENT
- DPCREDENTIALS
- DPLAPPINFO
- DPLCONNECTION
- DPNAME
- DPSECURITYDESC
- DPSESSIONDESC2

DPACCOUNTDESC

Describes the account information for a specific player.

```
typedef struct {
  DWORD  dwSize;
  DWORD  dwFlags;
  union {
    LPWSTR  lpszAccountID;
    LPSTR   lpszAccountIDA;
  };
} DPACCOUNTDESC, FAR *LPDPACCOUNTDESC;
```

Members

dwSize
The size of the DPACCOUNTDESC structure, *dwsize* = sizeof(DPACCOUNTDESC).

dwFlags
Not used. Must be zero.

lpszAccountID
Pointer to a Unicode string containing the account identifier. This is a unique identifier that describes a player who is securely logged in. The format of the identifier depends on the Security Support Provider Interface (SSPI) package being used.

lpszAccountIDA
Pointer to an ANSI string containing the account identifier. This is a unique identifier that describes a player who is securely logged in. The format of the identifier depends on the SSPI package being used.

See Also

IDirectPlay3::GetPlayerAccount

DPCAPS

Contains the capabilities of a DirectPlay object after a call to the IDirectPlay3::GetCaps or IDirectPlay3::GetPlayerCaps methods. Any of these capabilities can differ depending on whether guaranteed or nonguaranteed capabilities are requested. This structure is read-only.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwMaxBufferSize;
    DWORD dwMaxQueueSize;
    DWORD dwMaxPlayers;
    DWORD dwHundredBaud;
    DWORD dwLatency;
    DWORD dwMaxLocalPlayers;
    DWORD dwHeaderLength;
    DWORD dwTimeout;
} DPCAPS, FAR *LPDPCAPS;
```

Members

dwSize

The size of the DPCAPS structure, *dwsize* = sizeof(DPCAPS). Your application must set this member before it uses this structure; otherwise, an error will result.

dwFlags

Indicates the properties of the DirectPlay object.

DPCAPS_ENCRYPTIONSUPPORTED

Indicates that message encryption is supported by this DirectPlay object, either because it is a secure session or because the service provider can encrypt messages.

DPCAPS_GROUPOPTIMIZED

Indicates that the service provider bound to this DirectPlay object can optimize group (multicast) messaging.

DPCAPS_GUARANTEEDOPTIMIZED

Indicates that the service provider bound to this DirectPlay object supports guaranteed message delivery.

DPCAPS_GUARANTEEDSUPPORTED

Indicates that the DirectPlay object supports guaranteed message delivery, either because the service provider supports it or because DirectPlay implements it on a nonguaranteed service provider.

DPCAPS_ISHOST

Indicates that the DirectPlay object created by the calling application is the session host.

DPCAPS_KEEPALIVEOPTIMIZED

The service provider can detect when the connection to the session has been lost.

DPCAPS_SIGNINGSUPPORTED

Indicates that message authentication is supported by this DirectPlay object , either because it is a secure session or because the service provider can sign messages.

dwMaxBufferSize

Maximum number of bytes that can be sent in a single packet by this service provider. Larger messages will be sent by using more than one packet.

dwMaxQueueSize

This member is no longer used.

dwMaxPlayers

Maximum number of local and remote players supported in a session by this DirectPlay object.

dwHundredBaud

Bandwidth specified in multiples of 100 bits per second. For example, a value of 24 specifies 2400

bits per second. .

dwLatency

Estimate of latency by the 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. Latency can differ depending on whether the application uses guaranteed or nonguaranteed message delivery.

dwMaxLocalPlayers

Maximum number of local players supported in a session.

dwHeaderLength

Size, in bytes, of the header that will be added to player messages by this DirectPlay object. Note that the header size depends on which service provider is in use.

dwTimeout

Service provider's suggested time-out value. By default, DirectPlay will use this time-out value when waiting for replies to messages.

See Also

IDirectPlay3::Send

DPCHAT

Contains a DirectPlay chat message. Chat messages are sent with the IDirectPlay3::SendChatMessage method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    union{
      LPWSTR lpszMessage;
      LPSTR  lpszMessageA;
    };
} DPCHAT, FAR *LPDPCHAT;
```

Members

dwSize
The size of the DPCHAT structure, *dwsize* = sizeof(DPCHAT).

dwFlags
Not used. Must be 0.

lpszMessage
Pointer to a Unicode string containing the message to be sent. Can only be used with a Unicode interface (IDirectPlay3).

lpszMessageA
Pointer to an ANSI string containing the message to be sent. Can only be used with an ANSI interface (IDirectPlay3A).

See Also

IDirectPlay3::SendChatMessage

DPCOMPORTADDRESS

Contains information about the configuration of the COM port.

```
typedef struct DPCOMPORTADDRESS{
    DWORD dwComPort;
    DWORD dwBaudRate;
    DWORD dwStopBits;
    DWORD dwParity;
    DWORD dwFlowControl;
} DPCOMPORTADDRESS;

typedef DPCOMPORTADDRESS FAR* LPDPCOMPORTADDRESS;
```

Members

dwComPort

Indicates the number of the COM port to use. The value for this member can be 1, 2, 3, or 4.

dwBaudRate

Indicates the baud of the COM port. The value for this member can be one of the following:

CBR_110	CBR_300	CBR_600
CBR_1200	CBR_2400	CBR_4800
CBR_9600	CBR_14400	CBR_19200
CBR_38400	CBR_56000	CBR_57600
CBR_115200	CBR_128000	CBR_256000

dwStopBits

Indicates the number of stop bits. The value for this member can be ONESTOPBIT, ONE5STOPBITS, or TWOSTOPBITS.

dwParity

Indicates the parity used on the COM port. The value for this member can be NOPARITY, ODDPARITY, EVENPARITY, or MARKPARITY.

dwFlowControl

Indicates the method of flow control used on the COM port. The following values can be used for this member:

DPCPA_DTRFLOW	Indicates hardware flow control with DTR.
DPCPA_NOFLOW	Indicates no flow control.
DPCPA_RTSDTRFLOW	Indicates hardware flow control with RTS and DTR.
DPCPA_RTSFLOW	Indicates hardware flow control with RTS.
DPCPA_XONXOFFFLOW	Indicates software flow control (xon/xoff).

Remarks

The constants that define baud, stop bits, and parity are defined in Winbase.h.

DPCOMPOUNDADDRESSELEMENT

Describes a DirectPlay Address data chunk. See DirectPlay Address Data Types for a list of predefined Microsoft data types.

```
typedef struct {
    GUID    guidDataType;
    DWORD   dwDataSize;
    LPVOID  lpData;
} DPCOMPOUNDADDRESSELEMENT, FAR *LPDPCOMPOUNDADDRESSELEMENT;
```

Members

guidDataType
GUID identifying the type of data contained in this structure.

dwDataSize
Size of the data in bytes.

lpData
Pointer to a buffer containing the data.

See Also

IDirectPlayLobby2::CreateCompoundAddress

DPCREDENTIALS

Holds the user name, password, and domain to connect to a secure server.

```
typedef struct {
    DWORD  dwSize;
    DWORD  dwFlags;
    union {
           LPWSTR lpszUsername;
           LPSTR  lpszUsernameA;
          };
    union {
           LPWSTR lpszPassword;
           LPSTR  lpszPasswordA;
          };
    union {
           LPWSTR lpszDomain;
           LPSTR  lpszDomainA;
          };

} DPCREDENTIALS, FAR *LPDPCREDENTIALS;
```

Members

dwSize
The size of the **DPCREDENTIALS** structure, *dwsize* = sizeof(DPCREDENTIALS).

dwFlags
Not used. Must be zero.

lpszUsername, **lpszPassword**, **lpszDomain**
Pointers to Unicode strings containing the user name, password, and domain name. Can only be used with a Unicode interface.

lpszUsernameA, **lpszPasswordA**, **lpszDomainA**
Pointers to ANSI strings containing the user name, password, and domain name. Can only be used with an ANSI interface.

See Also

IDirectPlay3::SecureOpen

DPLAPPINFO

Contains information about the application from the registry and is passed to the IDirectPlayLobby2::EnumLocalApplications callback function.

```
typedef struct {
    DWORD  dwSize;
    GUID   guidApplication;
    union {
           LPSTR  lpszAppNameA;
           LPWSTR lpszAppName;
          };
} DPLAPPINFO, FAR *LPDPLAPPINFO;
```

Members

dwSize

The size of the DPLAPPINFO structure, *dwsize* = sizeof(DPLAPPINFO). Your application must set this member before it uses this structure; otherwise, an error will result.

guidApplication

Globally unique identifier (GUID) of the application.

lpszAppNameA, **lpszAppName**

Name of the application in ANSI or Unicode, depending on what interface is in use.

DPLCONNECTION

Contains the information needed to connect an application to a session.

```
typedef struct {
    DWORD            dwSize;
    DWORD            dwFlags;
    LPDPSESSIONDESC2 lpSessionDesc;
    LPDPNAME         lpPlayerName;
    GUID             guidSP;
    LPVOID           lpAddress;
    DWORD            dwAddressSize;
} DPLCONNECTION, FAR *LPDPLCONNECTION;
```

Members

dwSize

The size of the DPLCONNECTION structure, *dwsize* = sizeof(DPLCONNECTION). Your application must set this member before it uses this structure; otherwise, an error will result. Examine this member to determine if the fields added in DirectPlay version 5 are available.

dwFlags

Indicates how to open a session.

DPLCONNECTION_CREATESESSION

Create a new session as described in the session description.

DPLCONNECTION_JOINSESSION

Join the existing session as described in the session description.

lpSessionDesc

Pointer to a DPSESSIONDESC2 structure describing the session to be created or the session to join.

lpPlayerName

Pointer to a DPNAME structure holding the name the player should be created with. This will be the name of the person registered in the lobby. The application can ignore this name.

guidSP

Globally unique identifier (GUID) of the service provider to use to connect to the session.

lpAddress

Pointer to a DirectPlay Address that contains the information that the service provider needs to connect to a session. For more information about the DirectPlay Address, see DirectPlay Address. For a list of Microsoft predefined address data types, see DirectPlay Address Data Types.

dwAddressSize

Size, in bytes, of the address data.

See Also

IDirectPlayLobby2::RunApplication, IDirectPlayLobby2::GetConnectionSettings, IDirectPlayLobby2::SetConnectionSettings

DPNAME

Contains name information for a DirectPlay entity, such as a player or group.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    union {
        LPWSTR lpszShortName;
        LPSTR  lpszShortNameA;
    };
    union {
        LPWSTR lpszLongName;
        LPSTR  lpszLongNameA;
    };
} DPNAME, FAR *LPDPNAME;
```

Members

dwSize

The size of the DPNAME structure, *dwsize* = sizeof(DPNAME). Your application must set this member before it uses this structure; otherwise, an error will result.

dwFlags

Structure-specific flags. Currently set to zero.

lpszShortName and **lpszLongName**

Pointers to Unicode strings containing the short (friendly) and long (formal) names of a player or group. Use these members only if the IDirectPlay3 interface is in use.

lpszShortNameA and **lpszLongNameA**

Pointers to ANSI strings containing the short (friendly) and long (formal) names of a player or group. Use these members only if the IDirectPlay3A interface is in use.

See Also

IDirectPlay3::CreateGroup, IDirectPlay3::CreatePlayer, IDirectPlay3::GetGroupName, IDirectPlay3::GetPlayerName, IDirectPlay3::SetGroupName, IDirectPlay3::SetPlayerName

DPSECURITYDESC

Describes the security properties of a DirectPlay session instance.

```
typedef struct {
    DWORD  dwSize;
    DWORD  dwFlags;
    union {
           LPWSTR  lpszSSPIProvider;
           LPSTR   lpszSSPIProviderA;
    };
    union {
           LPWSTR lpszCAPIProvider;
           LPSTR  lpszCAPIProviderA;
    };
    DWORD  dwCAPIProviderType;
    DWORD  dwEncryptionAlgorithm;
} DPSECURITYDESC, FAR *LPDPSECURITYDESC;
```

Members

dwSize
The size of the DPSECURITYDESC structure, *dwsize* = sizeof(DPSECURITYDESC).

dwFlags
Not used. Must be zero.

lpszSSPIProvider, **lpszSSPIProviderA**
Pointer to a Unicode or ANSI string describing the Security Support Provider Interface (SSPI) package to use for authenticated logins. Pass NULL to use the default, the NTLM (NT LAN Manager) security provider.

lpszCAPIProvider, **lpszCAPIProviderA**
Pointer to a Unicode or ANSI string describing the CryptoAPI package to use for cryptography services. Pass NULL to use the default, the Microsoft RSA Base Cryptographic Provider v. 1.0.

dwCAPIProviderType
CryptoAPI service provider type. Pass zero to use the default type, PROV_RSA_FULL.

dwEncryptionAlgorithm
Encryption algorithm to use. DirectPlay only supports stream ciphers. Pass zero to use the default, the CALG_RC4 stream cipher.

Remarks

For more information about the CryptoAPI, see the CrpytoAPI and Cryptography topics at http://www.microsoft.com. The Microsoft RSA Base Cryptographic Provider is supplied by Microsoft and is included with the Windows 95 and Windows NT operating systems.

DirectPlay does not support block encryption.

See Also

IDirectPlay3::SecureOpen

DPSESSIONDESC2

Contains a description of an IDirectPlay3 session's capabilities. (The **DPSESSIONDESC** structure is no longer used in the IDirectPlay3 interface.)

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    GUID  guidInstance;
    GUID  guidApplication;
    DWORD dwMaxPlayers;
    DWORD dwCurrentPlayers;
    union  {
        LPWSTR lpszSessionName;
        LPSTR  lpszSessionNameA;
    };
    union  {
        LPWSTR lpszPassword;
        LPSTR  lpszPasswordA;
    };
    DWORD dwReserved1;
    DWORD dwReserved2;
    DWORD dwUser1;
    DWORD dwUser2;
    DWORD dwUser3;
    DWORD dwUser4;
} DPSESSIONDESC2, FAR *LPDPSESSIONDESC2;
```

Members

dwSize

The size of the DPSESSIONDESC2 structure, *dwsize* = sizeof(DPSESSIONDESC2). Your application must set this member before it uses this structure; otherwise, an error will result.

dwFlags

A combination of the following flags. These flags can be changed after a session has started by using IDirectPlay3::SetSessionDesc.

DPSESSION_CLIENTSERVER

This is a client/server session that is being hosted by an application server process. This flag must be specified at the time the session is created. Any clients that join this session will only be able to enumerate the server player and any local players. The host of the session will see all the players. If this flag is not set, the session is a peer-to-peer session. If this flag is set, the host can only create a server player. This flag cannot be used with DPSESSION_MIGRATEHOST.

DPSESSION_JOINDISABLED

No new applications can join this session. Any call to the IDirectPlay3::Open method with the DPOPEN_JOIN flag and the globally unique identifier (GUID) of this session instance will cause an error. If this flag is not specified, new remote applications can join the session until the session player limit is reached.

DPSESSION_KEEPALIVE

Automatically detect when remote players drop out of the game abnormally. Those players will be deleted from the session. If a temporary network outage caused the loss of the players, they will be informed when they return that they were dropped from the session through the DPMSG_SESSIONLOST system message. This flag must be specified at the time the session is created. If this flag is not specified, DirectPlay will not automatically keep the session alive if players are abnormally terminated.

DPSESSION_MIGRATEHOST

If the current host exits, another computer in the session will become the host. The players on the new host computer will receive a DPMSG_HOST system message. This flag must be

specified at the time the session is created. If this flag is not specified, the host will not migrate, new computers cannot join the session, and new players cannot be created if the current host leaves. Note that the DPSESSION_MIGRATEHOST flag cannot be used with the DPSESSION_CLIENTSERVER, DPSESSION_MULTICASTSERVER, or DPSESSION_SECURESERVER flags.

DPSESSION_MULTICASTSERVER

In a peer-to-peer session, broadcast and group messages are routed through the host which acts like a multicast server. This flag must be specified at the time the session is created. The flag is only useful if the host has a high bandwidth connection to the network. If this flag is not specified, broadcast and group messages are sent directly between peers. This flag cannot be used with DPSESSION_MIGRATEHOST.

DPSESSION_NEWPLAYERSDISABLED

Indicates that new players cannot be created in the session. Any call to the IDirectPlay3::CreatePlayer method by an application in the session will result in an error. Also, new applications cannot join the session. If this flag is not specified, players can be created until the session player limit is reached.

DPSESSION_NODATAMESSAGES

Do not send system messages when remote player data, group data, or session data is changed with the IDirectPlay3::SetPlayerData, IDirectPlay3::SetGroupData, IDirectPlay3::SetPlayerName, IDirectPlay3::SetGroupName, or IDirectPlay3::SetSessionDesc method. If this flag is not specified, messages will be generated indicating that the data changed.

Note that setting this flag also suppresses the DPMSG_SETSESSIONDESC message.

DPSESSION_NOMESSAGEID

Do not attach data to messages indicating what player the message is from and to whom it is sent. Saves message overhead if this information is not relevant. (For more information, see the IDirectPlay3::Receive method.) If this flag is not specified, the message ID will be added. This flag must be specified at the time the session is created.

DPSESSION_PASSWORDREQUIRED

This session is password protected. Any applications wishing to join the session must supply the password in the IDirectPlay3::Open call. This is a read-only flag. It will be set automatically by DirectPlay when the host of the session specifies a non-NULL password.

DPSESSION_PRIVATE

This is a private session. It can not respond to enumeration requests unless the request contains a non-NULL, matching password. If this flag is not specified, the session will respond to enumeration requests.

DPSESSION_SECURESERVER

This session is being hosted on a secure server which will require authentication credentials before it can be opened. The host must specify this flag when the session is created. It cannot be changed later through IDirectPlay3::SetSessionDesc. This flag can only be used if the DPSESSION_CLIENTSERVER flag is also specified. This flag cannot be used with the DPSESSION_MIGRATEHOST flag.

guidInstance

GUID of the session instance.

guidApplication

GUID for the application running in the session instance. It uniquely identifies the application so that DirectPlay connects only to other computers running the same application. This member can be set to GUID_NULL to enumerate sessions for any application.

dwMaxPlayers

Maximum number of players allowed in this session. A value of zero means there is no maximum.

dwCurrentPlayers

Number of players currently in the session.

lpszSessionName

Pointer to a Unicode string containing the name of the session. Use only if a Unicode DirectPlay interface (IDirectPlay3) is being used.

lpszSessionNameA

Pointer to an ANSI string containing the name of the session. Use only if an ANSI DirectPlay interface (IDirectPlay3A) is being used.

lpszPassword

Pointer to a Unicode string containing the password used to join the session for participation. Use only if a Unicode DirectPlay interface (IDirectPlay3) is being used.

lpszPasswordA

Pointer to an ANSI string containing the password used to join the session for participation. Use only if an ANSI DirectPlay interface (IDirectPlay3A) is being used.

dwReserved1 and **dwReserved2**

Must be zero.

dwUser1, **dwUser2**, **dwUser3**, and **dwUser4**

Application-specific data for the session.

See Also

IDirectPlay3::EnumSessions, IDirectPlay3::GetSessionDesc, IDirectPlay3::Open, IDirectPlay3::SecureOpen, IDirectPlay3::SetSessionDesc, IDirectPlay3::CreatePlayer, DPPLAYER_SERVERPLAYER

System Messages

Some system message structures have changed for DirectPlay 5, with new data members added to the end. Make sure you only reference these new members if you are using the IDirectPlay3 interface.

The system message structures that have new members for DirectPlay 5 are:

Structure	New Members
DPMSG_CREATEPLAYERORGROUP	DPID dpIdParent
	DWORD dwFlags
DPMSG_DESTROYPLAYERORGROUP	DPNAME dpnName
	DPID dpIdParent
	DWORD dwFlags

DPMSG_ADDGROUPTOGROUP

DirectPlay generates this message and sends it to every player when a group is added to a group.

```
typedef struct{
    DWORD    dwType;
    DPID     dpIdParentGroup;
    DPID     dpIdGroup;
} DPMSG_ADDGROUPTOGROUP, FAR *LPDPMSG_ADDGROUPTOGROUP;
```

Members

dwType
Identifies the message. This member is DPSYS_ADDGROUPTOGROUP.

dpIdParentGroup
ID of the group to which the group was added.

dpIdGroup
ID of the group that was added to the group.

See Also

IDirectPlay3::AddGroupToGroup

DPMSG_ADDPLAYERTOGROUP

Contains information for the DPSYS_ADDPLAYERTOGROUP system message. DirectPlay generates this message and sends it to every player when a player is added to a group.

The application can use IDirectPlay3::GetPlayerCaps, IDirectPlay3::GetPlayerName, and IDirectPlay3::GetPlayerData for information about the player involved int his message, or IDirectPlay3::GetGroupName and IDirectPlay3::GetGroupData to get more information about the group involved in this message.

```
typedef struct{
    DWORD dwType;
    DPID  dpIdGroup;
    DPID  dpIdPlayer;
} DPMSG_ADDPLAYERTOGROUP, FAR *LPDPMSG_ADDPLAYERTOGROUP;
```

Members

dwType
Identifies the message. This member is DPSYS_ADDPLAYERTOGROUP.

dpIdGroup
ID of the group to which the player was added.

dpIdPlayer
ID of the player that was added to the group.

See Also

IDirectPlay3::AddPlayerToGroup

DPMSG_CHAT

Contains information for the DPSYS_CHAT system message. The system message is generated for a local player receiving a chat message from another player.

```
typedef struct{
    DWORD dwType;
    DWORD dwFlags;
    DPID  idFromPlayer;
    DPID  idToPlayer;
    DPID  idToGroup;
    LPDPCHAT  lpChat;
} DPMSG_CHAT, FAR *LPDPMSG_CHAT;
```

Members

dwType
Identifies the message. This member has the value DPSYS_CHAT.

dwFlags
Not used.

idFromPlayer
The DPID of the player from whom the message originated. DPID_SERVERPLAYER indicates the message originated from the server.

idToPlayer
The DPID of the player to whom the message was directed. If this DPID is zero, then the message was sent to a group or broadcast to everyone.

idToGroup
The DPID of the group to whom the message was directed. If this DPID is zero and *idToPlayer* is also zero, then the message was broadcast to everyone.

lpChat
Pointer to a DPCHAT structure containing the content of the chat message received.

Remarks

Use the *idFromPlayer* value to determine who sent the message. The value of *lpidFrom* available through Receive will always be zero, so you must retrieve the value of *idFromPlayer* in the **DPMSG_CHAT** structure.

See Also

IDirectPlay3::SendChatMessage, DPCHAT, IDirectPlay3::Receive

DPMSG_CREATEPLAYERORGROUP

Contains information for the DPSYS_CREATEPLAYERORGROUP system message. The system sends this message when players and groups are created in a session.

DirectPlay generates this message and sends it to each player when a new player or group is created in a session.

```
typedef struct{
    DWORD  dwType;
    DWORD  dwPlayerType;
    DPID   dpId;
    DWORD  dwCurrentPlayers;
    LPVOID lpData;
    DWORD  dwDataSize;
    DPNAME dpnName;
    DPID   dpIDParent;
    DWORD  dwFlags;
} DPMSG_CREATEPLAYERORGROUP, FAR *LPDPMSG_CREATEPLAYERORGROUP;
```

Members

dwType
Identifies the message. This member must be set to DPSYS_CREATEPLAYERORGROUP.

dwPlayerType
Indicates whether the message applies to a player (DPPLAYERTYPE_PLAYER) or a group (DPPLAYERTYPE_GROUP).

dpId
The ID of a player or group created.

dwCurrentPlayers
Current number of players in the session before this player was created.

lpData
Pointer to any application-specific remote data associated with this player or group. If this member is NULL, there is no remote data.

dwDataSize
Size of the data contained in the buffer referenced by *lpData*.

dpnName
Structure containing the name of the player or group.

dpIdParent
The ID of the parent group if this message is caused by a call to IDirectPlay3::CreateGroupInGroup; otherwise, the value is 0.

dwFlags
The player or group flags.

Remarks

The DirectPlay 5 version of this structure has two members added to the end, *dpIdParent* and *dwFlags*.

See Also

IDirectPlay3::CreateGroup, IDirectPlay3::CreatePlayer, IDirectPlay3::CreateGroupInGroup

DPMSG_DELETEGROUPFROMGROUP

DirectPlay generates this message and sends it to every player when a group is removed from a group. Note that this structure is identical to DPMSG_ADDGROUPTOGROUP.

```
typedef struct{
    DWORD    dwType;
    DPID     dpIdParentGroup;
    DPID     dpIdGroup;
} DPMSG_ADDGROUPTOGROUP, FAR *LPDPMSG_ADDGROUPTOGROUP;
typedef DPMSG_ADDGROUPTOGROUP  DPMSG_DELETEGROUPFROMGROUP;
```

Members

dwType
Identifies the message. This member is DPSYS_DELETEGROUPFROMGROUP.

dpIdParentGroup
ID of the group from which the group was removed.

dpIdGroup
ID of the group that was removed from the group.

See Also

IDirectPlay3::DeleteGroupFromGroup

DPMSG_DELETEPLAYERFROMGROUP

Contains information for the DPSYS_DELETEPLAYERFROMGROUP system message. DirectPlay generates this message and sends it to each local player on the computer when a player is deleted from a group.

For a description of the structure members, see the DPMSG_ADDPLAYERTOGROUP structure.

The application can use IDirectPlay3::GetPlayerCaps, IDirectPlay3::GetPlayerName, and IDirectPlay3::GetPlayerData for information about the player involved int his message, or IDirectPlay3::GetGroupName and IDirectPlay3::GetGroupData to get more information about the group involved in this message.

```
typedef struct{
    DWORD dwType;
    DPID  dpIdGroup;
    DPID  dpIdPlayer;
} DPMSG_ADDPLAYERTOGROUP, FAR *LPDPMSG_ADDPLAYERTOGROUP;
typedef DPMSG_ADDPLAYERTOGROUP DPMSG_DELETEPLAYERFROMGROUP;
```

Members

dwType
Identifies the message. This member is DPSYS_DELETEPLAYERFROMGROUP.

dpIdGroup
ID of the group from which the player was removed.

dpIdPlayer
ID of the player that was removed from the group.

See Also

IDirectPlay3::DeletePlayerFromGroup

DPMSG_DESTROYPLAYERORGROUP

Contains information for the DPSYS_DESTROYPLAYERORGROUP system message. DirectPlay generates this message and sends it to each player when a player or group is destroyed in a session.

```
typedef struct{
    DWORD  dwType;
    DWORD  dwPlayerType;
    DPID   dpId;
    LPVOID lpLocalData;
    DWORD  dwLocalDataSize;
    LPVOID lpRemoteData;
    DWORD  dwRemoteDataSize;
    DPNAME dpnName;
    DPID   dpIdParent;
    DWORD  dwFlags;
} DPMSG_DESTROYPLAYERORGROUP, FAR *LPDPMSG_DESTROYPLAYERORGROUP;
```

Members

dwType
Identifies the message. This member is DPSYS_DESTROYPLAYERORGROUP.

dwPlayerType
Identifies whether the message applies to a player (DPPLAYERTYPE_PLAYER) or group (DPPLAYERTYPE_GROUP).

dpId
ID of a player or group that has been destroyed.

lpLocalData
Pointer to the local data associated with this player/group.

dwLocalDataSize
Size, in bytes, of the local data.

lpRemoteData
Pointer to the remote data associated with this player/group.

dwRemoteDataSize
Size, in bytes, of the remote data.

dpnName
Structure containing the name of the player/group.

dpIdParent
The ID of the parent group if the group being destroyed is a subgroup of the parent group (the group being destroyed was created by a call to IDirectPlay3::CreateGroupInGroup; otherwise, the value is 0.

dwFlags
The player or group flags.

Remarks

The DirectPlay 5 version of this structure has three members added to the end, *dpnName*, *dpIdParent* and *dwFlags*.

See Also

IDirectPlay3::DestroyGroup, IDirectPlay3::DestroyPlayer

DPMSG_GENERIC

This structure is provided for message processing.

```
typedef struct{
    DWORD dwType;
} DPMSG_GENERIC, FAR *LPDPMSG_GENERIC;
```

Members

dwType
Identifies the system message type.

Remarks

When a system message is received (that is, the value pointed to by the *lpidFrom* parameter equals DPID_SYSMSG), first cast the unknown message data to the **DPMSG_GENERIC** type, and then perform further processing based on the value of **dwType**. After the message type has been determined, the message can cast to one of the known types of system messages for further processing.

DPMSG_HOST

When the current session host exits the session, this message is sent to all the players on the computer that inherits the host duties.

```
typedef DPMSG_GENERIC    DPMSG_HOST;
typedef DPMSG_HOST  FAR *LPDPMSG_HOST;
```

DPMSG_SECUREMESSAGE

DirectPlay generates this message when it receives a signed or encrypted message from another player.

```
typedef struct {
    DWORD   dwType;
    DWORD   dwFlags;
    DPID    dpIdFrom;
    LPVOID  lpData;
    DWORD   dwDataSize;
} DPMSG_SECUREMESSAGE, FAR *LPDPMSG_SECUREMESSAGE;
```

Members

dwType
Identifies the system message type. This is DPSYS_SECUREMESSAGE.

dwFlags
Flags indicating how the message was secured by the sender. One of the following values:

DPSEND_SIGNED - the message was signed by the sender and the signature was successfully verified.

DPSEND_ENCRYPTED - the messages was encrypted by the sender and successfully decrypted.

dpIdFrom
The DPID of the player that sent the secure message.

lpData
Pointer to a buffer containing the fully verified message.

dwDataSize
Size of the buffer containing the message.

See Also

IDirectPlay3::Send

DPMSG_SESSIONLOST

This message is generated by DirectPlay when the connection to all the other players in the session is lost. After the session is lost, messages cannot be sent to remote players, but all data at the time the session was lost is still available. Your applications should try to recover gracefully and exit if this message is received.

```
typedef DPMSG_GENERIC    DPMSG_SESSIONLOST;
typedef DPMSG_SESSIONLOST  FAR  *LPDPMSG_SESSIONLOST;
```

DPMSG_SETPLAYERORGROUPDATA

Contains information for the DPSYS_SETPLAYERORGROUPDATA system message. DirectPlay generates this message and sends it to each player when the remote data of a player or group changes. This message will not be generated if the DPSESSION_ NODATAMESSAGES flag is specified in the session description.

```
typedef struct {
    DWORD  dwType;
    DWORD  dwPlayerType;
    DPID   dpId;
    LPVOID lpData;
    DWORD  dwDataSize;
} DPMSG_SETPLAYERORGROUPDATA, FAR *LPDPMSG_SETPLAYERORGROUPDATA;
```

Members

dwType
Identifies the message. This member is always DPSYS_SETPLAYERORGROUPDATA.

dwPlayerType
Identifies whether the message applies to a player (DPPLAYERTYPE_PLAYER) or a group (DPPLAYERTYPE_GROUP).

dpId
ID of the player or group whose data changed.

lpData
Pointer to an application-specific block of data.

dwDataSize
Size of the data contained in the buffer referenced by **lpData**.

Remarks

It is not necessary for the application to save the data from this message because it can be retrieved at any time using IDirectPlay3::GetPlayerData or IDirectPlay3::GetGroupData.

See Also

IDirectPlay3::GetPlayerData, IDirectPlay3::GetGroupData, IDirectPlay3::SetPlayerData, IDirectPlay3::SetGroupData

DPMSG_SETPLAYERORGROUPNAME

Contains information for the DPSYS_SETPLAYERORGROUPNAME system message. DirectPlay generates this message and sends it to each local player on the computer when the name of a player or group changes. This message will not be generated if the DPSESSION_ NODATAMESSAGES flag is specified in the session description.

```
typedef struct {
    DWORD  dwType;
    DWORD  dwPlayerType;
    DPID   dpId;
    DPNAME dpnName;
} DPMSG_SETPLAYERORGROUPNAME, FAR *LPDPMSG_SETPLAYERORGROUPNAME;
```

Members

dwType
Identifies the message. This member is always DPSYS_SETPLAYERORGROUPNAME.

dwPlayerType
Identifies whether the message applies to a player (DPPLAYERTYPE_PLAYER) or a group (DPPLAYERTYPE_GROUP).

dpId
ID of the player or group whose name changed.

dpnName
Structure containing the new name information for the player or group.

Remarks

It is not necessary for the application to save the *dpnName* from this message because it can be retrieved at any time using IDirectPlay3::GetPlayerName or IDirectPlay3::GetGroupName.

See Also

IDirectPlay3::GetPlayerName, IDirectPlay3::GetGroupName, IDirectPlay3::SetPlayerName, IDirectPlay3::SetGroupName

DPMSG_SETSESSIONDESC

Contains information for the DPSYS_SETSESSIONDESC system message. Every player will receive this system message when the session description changes.

This messages will not be generated if the DPSESSION_NODATAMESSAGES flag is specified in the session description.

```
typedef struct
{
    DWORD           dwType;
    DPSESSIONDESC2  dpDesc;
} DPMSG_SETSESSIONDESC, FAR *LPDPMSG_SETSESSIONDESC;
```

Members

dwType
Identifies the message. This member is always DPSYS_SETSESSIONDESC.

dpDesc
Pointer to a DPSESSIONDESC2 structure containing the updated session description.

See Also

IDirectPlay3::GetSessionDesc, IDirectPlay3::SetSessionDesc

DPMSG_STARTSESSION

Contains information for the DPSYS_STARTSESSION system message. The lobby server sends this message to each player in a group when it is time for that player to join an application session.

```
typedef struct {
   DWORD            dwType;
   LPDPLCONNECTION  lpConn;
} DPMSG_STARTSESSION, FAR *LPDPMSG_STARTSESSION;
```

Members

dwType

Identifies the message. This member is always DPSYS_STARTSESSION.

lpConn

Pointer to a DPLCONNECTION structure that contains all the information needed to launch an application into a session. This structure can be passed into IDirectPlayLobby2::RunApplication (if an external application is to be launched) or passed into IDirectPlayLobby2::SetConnectionSettings and then followed by a call to IDirectPlayLobby2::Connect (to connect the current application to the session).

Remarks

Upon receipt of this message by a stand-alone lobby client, the client must launch the application by calling the IDirectPlayLobby2::RunApplication method with the DPLCONNECTION structure.

Upon receipt of this message by an application with an internal lobby, the application must set the connection settings by calling IDirectPlayLobby2::SetConnectionSettings and then connect to the session using the IDirectPlayLobby2::Connect method.

See Also

IDirectPlay3::StartSession, IDirectPlayLobby2::RunApplication, IDirectPlayLobby2::SetConnectionSettings, IDirectPlayLobby2::Connect

Standard Lobby Messages

To determine whether the lobby that launched your application supports standard lobby messages, it is necessary to send the lobby an initial DPLMSG_GETPROPERTY request and see whether it responds or not. The property that should be requested is DPLPROPERTY_MessagesSupported.

If the lobby responds with TRUE, then you can assume that it will respond to all further messages. If it responds with FALSE or doesn't respond at all within a specified timeout, then you can assume that messages are not supported.

If the application supports spectator players, then the application should also request DPLPROPERTY_PlayerGuid before creating players to determine if they should be created as spectators or not.

DPLMSG_GENERIC

Generic structure of system messages passed between the lobby client and an application.

```
typedef struct {
    DWORD dwType;
} DPL_GENERIC, FAR *LPDPLMSG_GENERIC;
```

Members

dwType

Identifies what type of system message has been received.

DPLSYS_APPTERMINATED

Indicates the application started by IDirectPlayLobby2::RunApplication has terminated.

DPLSYS_CONNECTIONSETTINGSREAD

Indicates the application started by the IDirectPlayLobby2::RunApplication method has read the connection settings.

DPLSYS_DPLAYCONNECTFAILED

Indicates the application started by IDirectPlayLobby2::RunApplication failed to connect to a session.

DPLSYS_DPLAYCONNECTSUCCEEDED

Indicates the application started by IDirectPlayLobby2::RunApplication has created a session and is ready for other applications to join or has successfully joined a session.

DPLMSG_GETPROPERTY

Message sent by an application to the lobby to request the current value of a property. These properties can be information such as the ranking of a player, a bitmap representing a player, or initial configuration information for the game or players that was done inside the lobby.

```
typedef struct {
    DWORD   dwType;
    DWORD   dwRequestID;
    GUID    guidPlayer;
    GUID    guidPropertyTag;
} DPLMSG_GETPROPERTY, FAR *LPDPLMSG_GETPROPERTY;
```

Members

dwType

Identifies the message. This value is DPSYS_ GETPROPERTY.

dwRequestID

An application generated ID to identify the request. When the lobby responds, it will be tagged with this request ID. The application can use the request ID to match responses to pending requests.

guidPlayer

GUID identifying the player that this property applies to (if applicable). If the property is not player-specific, this member should be set to GUID_NULL. The GUID for the player or players created by this application can be obtained from the lobby by requesting the DPLPROPERTY_PlayerGuid property.

guidPropertyTag

A GUID identifying the property that is being requested. The property can one of the predefined ones listed in the section DirectPlay Defined Properties or the application/lobby can define its own GUIDs for additional properties.

Remarks

Each property is identified by a GUID (defined by the application developer or the lobby developer). When a request for a property is made, the lobby responds with a DPLMSG_GETPROPERTYRESPONSE message. Even if the lobby cannot supply the information, it should respond with an error indicating the information is unavailable.

The application should not block waiting for a response and should have a way to time-out pending requests that haven't been fulfilled.

See Also

DPLMSG_GETPROPERTYRESPONSE

DPLMSG_GETPROPERTYRESPONSE

Message sent by a lobby to an application in response to a DPLMSG_GETPROPERTY message. The request that is being filled is identified by the *dwRequestID* parameter.

```
typedef struct {
    DWORD    dwType;
    DWORD    dwRequestID
    GUID     guidPlayer;
    GUID     guidPropertyTag;
    HRESULT  hr;
    DWORD    dwDataSize;
    DWORD    dwPropertyData[1];
} DPLMSG_GETPROPERTYRESPONSE, FAR *LPDPLMSG_GETPROPERTYRESPONSE;
```

Members

dwType

Identifies the message. This value is DPSYS_ GETPROPERTYRESPONSE.

dwRequestID

The ID that identifies the DPLMSG_GETPROPERTY message that this message is in response to.

guidPlayer

GUID identifying the player that this property applies to (if applicable). If the property is not player-specific, this member will be set to GUID_NULL. This will be the same as the GUID from the DPLMSG_GETPROPERTY message.

guidPropertyTag

A GUID identifying the property that is being requested. This will be the same as the GUID from the DPLMSG_GETPROPERTY message.

hr

Return code for the get property request. One of the following values:

DP_OK - successfully returned the property.

DPERR_UNKNOWN - the requested property is unknown to the lobby.

DPERR_UNAVAILABLE - the requested property is unavailable.

dwDataSize

The size, in bytes, of the property data.

dwPropertyData

A variable-size buffer that contains the property data. The property tag will define how to interpret this data.

Remarks

A lobby must either respond to all DPLMSG_GETPROPERTY requests or none of them.

When constructing this message, the lobby needs to allocate enough memory to hold the **DPLMSG_GETPROPERTYRESPONSE** structure and the complete property data. For example, if the property data requires 52 bytes, the lobby will allocate (sizeof(DPLMSG_GETPROPERTYRESPONSE) + 52) bytes and assign it to a **DPLMSG_GETPROPERTYRESPONSE** pointer.

See Also

DPLMSG_GETPROPERTY

DPLMSG_SETPROPERTY

Message sent by an application to the lobby to inform it that a property of a specific player or a property of the session has changed. These properties can range from the score or status of a player to the current level of a game or the current status of the session.

```
typedef struct {
    DWORD    dwType;
    DWORD    dwRequestID
    GUID     guidPlayer;
    GUID     guidPropertyTag;
    DWORD    dwDataSize;
    DWORD    dwPropertyData[1];
} DPLMSG_SETPROPERTY, FAR *LPDPLMSG_SETPROPERTY;
```

Members

dwType
Identifies the message. This value is DPSYS_ SETPROPERTY.

dwRequestID
A nonzero request ID supplied by the application if it would like confirmation that the data was recognized and set appropriately. If no confirmation is required, set this to DPL_NOCONFIRMATION.

guidPlayer
GUID identifying the player that this property applies to (if applicable). If the property is not player-specific, this member should be set to GUID_NULL. The GUID for the player or players created by this application can be obtained from the lobby by requesting the DPLPROPERTY_PlayerGuid property.

guidPropertyTag
A GUID identifying the property that is being set. The property can be one of the predefined GUIDs listed in the section DirectPlay Defined Properties or the application/lobby can define its own GUIDs for additional properties.

dwDataSize
The size, in bytes, of the property data.

dwPropertyData
A variable size buffer that contains the property data. The property tag will define how to interpret this data.

Remarks

Each property is identified by a GUID (defined by the application developer or the lobby developer), and it is the responsibility of the lobby to maintain a mapping of property GUIDs of the various applications to their descriptions and data types. The lobby server can choose to act on the information or ignore it.

When constructing this message, the application needs to allocate enough memory to hold the DPLMSG_ SETPROPERTY structure and the complete property data. For example, if the property data requires 52 bytes, the application will allocate (sizeof(DPLMSG_SETPROPERTY) + 52) bytes and assign it to a LPDPLMSG_SETPROPERTY pointer.

See Also

DPLMSG_SETPROPERTYRESPONSE

DPLMSG_SETPROPERTYRESPONSE

Message sent by a lobby to an application as confirmation of a DPLMSG_SETPROPERTY message. The message that is being confirmed is identified by the *dwRequestID* parameter.

```
typedef struct {
    DWORD    dwType;
    DWORD    dwRequestID
    GUID     guidPlayer;
    GUID     guidPropertyTag;
    HRESULT  hr;
} DPLMSG_SETPROPERTYRESPONSE, FAR *LPDPLMSG_SETPROPERTYRESPONSE;
```

Members

dwType

Identifies the message. This value is DPSYS_ SETPROPERTYRESPONSE.

dwRequestID

The ID that identifies the DPLMSG_SETPROPERTY message being confirmed.

guidPlayer

GUID identifying the player that this property was set for (if applicable). If the property is not player-specific, this member is GUID_NULL. This must be the same as the GUID from the DPLMSG_SETPROPERTY message.

guidPropertyTag

A GUID identifying the property that was set. This must be the same as the GUID from the DPLMSG_SETPROPERTY message.

hr

Result of the confirmation. One of the following values:

DP_OK - the property was set correctly.

DPERR_ACCESSDENIED - an attempt was made to set a property the lobby will not allow to be changed.

DPERR_UNKNOWN - the property is unknown to the lobby.

See Also

DPLMSG_SETPROPERTY

DirectPlay Defined Properties

This section describes the defined properties that can be set and retrieved by applications from a lobby.

When a DPLMSG_GETPROPERTYRESPONSE message is received, its *guidPropertyTag* field identifies the property and its *dwPropertyData* field contains the property information. To extract this information, cast the *dwPropertyData* field to the appropriate value. For example, the following routine retrieves the DPLDATA_PLAYERGUID structure from the given message:

```
BOOL GetPlayerGuid(LPDPLMSG_GETPROPERTYRESPONSE  lpPropertyResponseMsg,
                       LPDPLDATA_PLAYERGUID lpPlayerGuid)
{
   if (IsEqualGUID(lpPropertyResponseMsg->guidPropertyTag,
DPLPROPERTY_PlayerGuid))
   {
      *lpPlayerGuid = *((LPDPLDATA_PLAYERGUID) lpPropertyResponseMsg-
>dwPropertyData);
      return (TRUE);
   }
   else
   {
      return (FALSE);
   }
}
```

See the following descriptions of defined properties.

- DPLPROPERTY_LobbyGuid
- DPLPROPERTY_MessagesSupported
- DPLPROPERTY_PlayerGuid
- DPLPROPERTY_PlayerScore

DPLPROPERTY_LobbyGuid

A GUID used to identify the lobby software. If an application was designed to interoperate with a specific lobby using custom messages, this property can be used to identify the lobby.

However, an application should attempt to use standard messages whenever possible to be able to interoperate with as many lobbies as possible.

For this property, the *dwPropertyData* field of the DPLMSG_GETPROPERTYRESPONSE structure will contain the following:

```
GUID  guidLobby;
```

DPLPROPERTY_MessagesSupported

A boolean value indicating whether the lobby supports standard messages or not. An application can request this property from the lobby to determine if it supports standard messages. If the lobby does not respond to this message or if it responds with FALSE, then the application should assume that it does not support standard lobby messages and not send any further messages. If the lobby returns TRUE, then the application should freely send standard messages to the lobby and query for further properties of the lobby.

For this property, the *dwPropertyData* field of the DPLMSG_GETPROPERTYRESPONSE structure will contain the following:

```
BOOL  fMessagesSupported;
```

DPLPROPERTY_PlayerGuid

A GUID used to uniquely identify the application's local players. Any further DPLMSG_SETPROPERTY or DPLMSG_GETPROPERTY messages that require a player GUID will need to use this value to communicate with the lobby.

An application can manipulate properties for remote players on another computer. The application only needs to obtain the player GUID from the remote player (not from the lobby).

If an application supports spectator players, it should request this property before creating any players in order to determine what flags (if any) to use when creating the players; for example, DPPLAYER_SPECTATOR and DPPLAYER_SERVERPLAYER.

For this property, the *dwPropertyData* field of the DPLMSG_GETPROPERTYRESPONSE structure will contain the following:

```
typedef struct {
    GUID  guidPlayer;
    DWORD dwPlayerFlags;
} DPLDATA_PLAYERGUID, FAR *LPDPLDATA_PLAYERGUID;
```

DPLPROPERTY_PlayerScore

A generic structure that can be used by an application to report a composite score value to the lobby. The *guidPlayer* GUID of the player the score applies to must be provided when setting or getting this property. This structure can handle an arbitrary list of long integer values that collectively represent the score of a player. The application must allocate enough memory to hold all the scores. For example, if the score was passed as N integers, the application must allocate (sizeof(DPLDATA_PLAYERSCORE) + N*sizeof(LONG)) bytes and cast the pointer to LPDPLDATA_PLAYERSCORE.

For this property, the *dwPropertyData* field of the DPLMSG_GETPROPERTYRESPONSE structure will contain the following:

```
typedef struct {
    DWORD   dwScoreCount;
    LONG    Score[1];
} DPLDATA_PLAYERSCORE, FAR *LPDPLDATA_PLAYERSCORE;
```

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all IDirectPlay3 and IDirectPlayLobby2 methods. For a list of the error values each method can return, see the individual method descriptions.

CLASS_E_NOAGGREGATION

A non-NULL value was passed for the *pUnkOuter* parameter in DirectPlayCreate, DirectPlayLobbyCreate, or IDirectPlayLobby2::Connect.

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_APPNOTSTARTED

The application has not been started yet.

DPERR_AUTHENTICATIONFAILED

The password or credentials supplied could not be authenticated.

DPERR_BUFFERTOOLARGE

The data buffer is too large to store.

DPERR_BUSY

A message cannot be sent because the transmission medium is busy.

DPERR_BUFFERTOOSMALL

The supplied buffer is not large enough to contain the requested data.

DPERR_CANTADDPLAYER

The player cannot be added to the session.

DPERR_CANTCREATEGROUP

A new group cannot be created.

DPERR_CANTCREATEPLAYER

A new player cannot be created.

DPERR_CANTCREATEPROCESS

Cannot start the application.

DPERR_CANTCREATESESSION

A new session cannot be created.

DPERR_CANTLOADCAPI

No credentials were supplied and the CryptoAPI package (CAPI) to use for cryptography services cannot be loaded.

DPERR_CANTLOADSECURITYPACKAGE

The software security package cannot be loaded.

DPERR_CANTLOADSSPI

No credentials were supplied and the software security package (SSPI) that will prompt for credentials cannot be loaded.

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_CONNECTING

The method is in the process of connecting to the network. The application should keep calling the method until it returns DP_OK, indicating successful completion, or it returns a different error.

DPERR_ENCRYPTIONFAILED

The requested information could not be digitally encrypted. Encryption is used for message privacy. This error is only relevant in a secure session.

DPERR_EXCEPTION

An exception occurred when processing the request.

DPERR_GENERIC

An undefined error condition occurred.

DPERR_INVALIDCREDENTIALS

The credentials supplied (as to IDirectPlay3::SecureOpen) were not valid.

DPERR_INVALIDFLAGS

The flags passed to this method are invalid.

DPERR_INVALIDGROUP

The group ID is not recognized as a valid group ID for this game session.

DPERR_INVALIDINTERFACE

The interface parameter is invalid.

DPERR_INVALIDOBJECT

The DirectPlay object pointer is invalid.

DPERR_INVALIDPARAMS

One or more of the parameters passed to the method are invalid.

DPERR_INVALIDPASSWORD

An invalid password was supplied when attempting to join a session that requires a password.

DPERR_INVALIDPLAYER

The player ID is not recognized as a valid player ID for this game session.

DPERR_LOGONDENIED

The session could not be opened because credentials are required and either no credentials were supplied or the credentials were invalid.

DPERR_NOCAPS

The communication link that DirectPlay is attempting to use is not capable of this function.

DPERR_NOCONNECTION

No communication link was established.

DPERR_NOINTERFACE

The interface is not supported.

DPERR_NOMESSAGES

There are no messages in the receive queue.

DPERR_NONAMESERVERFOUND

No name server (host) could be found or created. A host must exist to create a player.

DPERR_NONEWPLAYERS

The session is not accepting any new players.

DPERR_NOPLAYERS

There are no active players in the session.

DPERR_NOSESSIONS

There are no existing sessions for this game.

DPERR_NOTLOBBIED

Returned by the IDirectPlayLobby2::Connect method if the application was not started by using the IDirectPlayLobby2::RunApplication method or if there is no DPLCONNECTION structure currently initialized for this DirectPlayLobby object.

DPERR_NOTLOGGEDIN

An action cannot be performed because a player or client application is not logged in. Returned by the IDirectPlay3::Send method when the client application tries to send a secure message without being logged in.

DPERR_OUTOFMEMORY

There is insufficient memory to perform the requested operation.

DPERR_PLAYERLOST

A player has lost the connection to the session.

DPERR_SENDTOOBIG

The message being sent by the IDirectPlay3::Send method is too large.

DPERR_SESSIONLOST

The connection to the session has been lost.

DPERR_SIGNFAILED

The requested information could not be digitally signed. Digital signatures are used to establish the authenticity of messages.

DPERR_TIMEOUT

The operation could not be completed in the specified time.

DPERR_UNAVAILABLE

The requested function is not available at this time.

DPERR_UNINITIALIZED

The requested object has not been initialized.

DPERR_UNKNOWNAPPLICATION

An unknown application was specified.

DPERR_UNSUPPORTED

The function is not available in this implementation. Returned from IDirectPlay3::GetGroupConnectionSettings and IDirectPlay3::SetGroupConnectionSettings if they are called from a session that is not a lobby session.

DPERR_USERCANCEL

Can be returned in two ways. 1) The user canceled the connection process during a call to the IDirectPlay3::Open method. 2) The user clicked Cancel in one of the DirectPlay service provider dialog boxes during a call to IDirectPlay3::EnumSessions.

Direct3D Immediate-Mode: Overview

This section provides overview information about the Direct3D® Immediate Mode. Information is divided into the following groups:

- About Direct3D Immediate Mode
- Why Use Direct3D Immediate Mode?
- Getting Started with Immediate Mode
- Direct3D Immediate-Mode Architecture
- Direct3D Immediate-Mode Essentials
- Direct3D Execute-Buffer Tutorial

About Direct3D Immediate Mode

Direct3D is designed to enable world-class game and interactive three-dimensional (3-D) graphics on a computer running Windows®. Its mission is to provide device-dependent access to 3-D video-display hardware in a device-independent manner. Simply put, Direct3D is a drawing interface for 3-D hardware.

You can use Direct3D in either of two modes: Immediate Mode or Retained Mode. Retained Mode is a high-level 3-D application programming interface (API) for programmers who require rapid development or who want the help of Retained Mode's built-in support for hierarchies and animation.

Microsoft developed the Direct3D Immediate Mode as a low-level 3-D API. 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 Retained Mode is built on top of Immediate Mode.

These are some of the advanced features of Direct3D:

- Switchable z-buffering
- Flat and Gouraud shading
- Phong lighting model, with multiple lights and light types
- Full material and texture support, including mipmapping
- Ramp and RGB software emulation
- Transformation and clipping
- Hardware independence
- Full support on NT
- Support for the Intel MMX architecture

Developers who use Immediate Mode instead of Retained Mode are typically experienced in high-performance programming issues, and may also be experienced in 3-D graphics. Your best source of information about Immediate Mode is probably the sample code included with this Software Development Kit (SDK); it illustrates how to put Direct3D Immediate Mode to work in real-world applications.

This section is not an introduction to programming with Direct3D Immediate Mode; for this information, see Direct3D Execute-Buffer Tutorial.

Why Use Direct3D Immediate Mode?

The world management of Immediate Mode is based on vertices, polygons, and commands that control them. It allows immediate access to the transformation, lighting, and rasterization 3-D graphics pipeline and provides emulation for missing hardware functionality. (The programmer is always told which capabilities are in hardware and which are being emulated.) Developers with existing 3-D applications and developers who need to achieve maximum performance by maintaining the thinnest possible layer between their application and the hardware should use Immediate Mode instead of Retained Mode.

There are two ways to use Immediate Mode: you can use the DrawPrimitive methods or you can work with execute buffers (display lists). Most developers who have never worked with Immediate Mode before will use the DrawPrimitive methods. Developers who already have an investment in code that uses execute buffers will probably continue to work with them. Neither technique is faster than the other – which you choose will depend on the needs of your application and your preferred programming style. For more information about these two ways to work with Immediate Mode, see The DrawPrimitive Methods and Using Execute Buffers.

Immediate Mode allows a low-overhead connection to 3-D hardware; there is no data translation between Direct3D and the Direct3D hardware-abstraction layer (HAL) for rendering operations. This low-overhead connection comes at a price; you must provide explicit calls for transformations and lighting, you must provide all the necessary matrices, and you must determine what kind of hardware is present and what its capabilities are.

Getting Started with Immediate Mode

The following sections describe some of the technical concepts you need to understand before you write programs that incorporate 3-D graphics. This is not a discussion of broad architectural details, nor is it an in-depth analysis of specific Direct3D components. (For information about these topics, see Direct3D Immediate-Mode Architecture and Direct3D Immediate-Mode Essentials.)

If you are already experienced in producing 3-D graphics, simply scan the following sections for information that is unique to Direct3D.

Information in this section is divided into the following groups:

- 3-D Coordinate Systems
- 3-D Transformations
- Polygons
- Triangle Strips and Fans
- Triangle Rasterization Rules

3-D Coordinate Systems

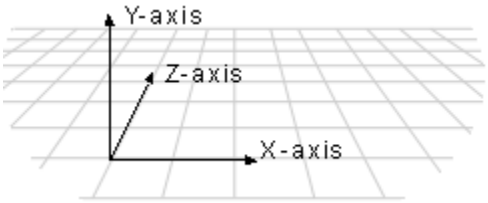
This section describes the Direct3D coordinate system and coordinate types that your application can use.

- Direct3D Coordinate System
- U- and V-Coordinates

Direct3D Coordinate System

There are two varieties of Cartesian coordinate systems in 3-D 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 uses a 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

In addition to the x-, y-, and z-coordinates that define space in a Cartesian coordinate system, Direct3D uses texture coordinates. These coordinates (u and v) define an "up" direction for the texture, typically along the y-axis (u) and an orientation along the plane of the texture, typically along the z-axis (v). As with all normalized vectors, the origins of these vectors are at [0,0,0].

For more information about texture coordinates, see Textures.

3-D Transformations

In programs that work with 3-D 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 4×4 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 this:

```
D3DMATRIX scale = {
    D3DVAL(s),    0,            0,            0,
    0,            D3DVAL(s),    D3DVAL(t),    0,
    0,            0,            D3DVAL(s),    D3DVAL(v),
    0,            0,            0,            D3DVAL(1)
};
```

This section describes the 3-D transformations available to your applications through Direct3D:

- Translation
- Rotation
- Scaling

For more information about transformations in Direct3D Immediate Mode, see Viewports and Transformations.

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

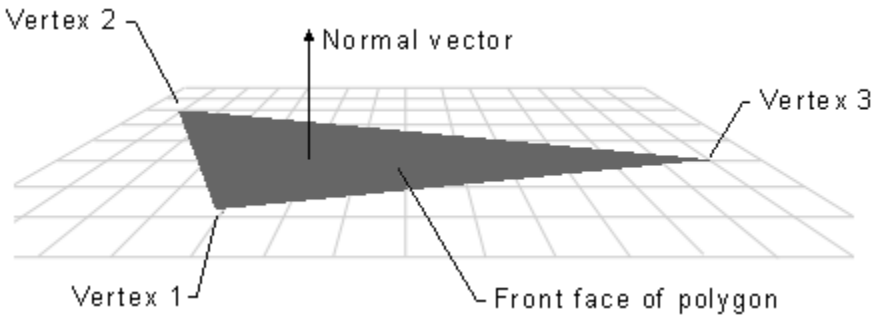
Three-dimensional objects in Direct3D are made up of meshes. A mesh is a set of faces, each of which is described by one or more triangles.

This section describes how your applications can use Direct3D polygons.

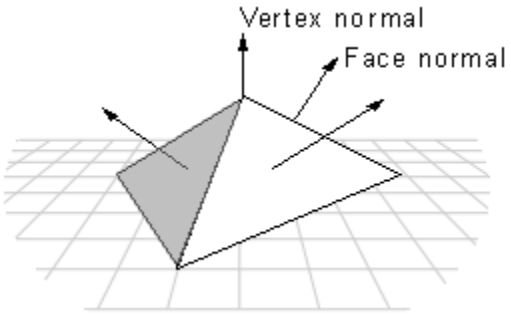
- Face and Vertex Normals
- Shade Modes
- Triangle Interpolants

Face and Vertex Normals

Each face in a mesh has a perpendicular normal vector 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 Gouraud shade modes, and for controlling lighting and texturing effects, the system uses vertex normals.

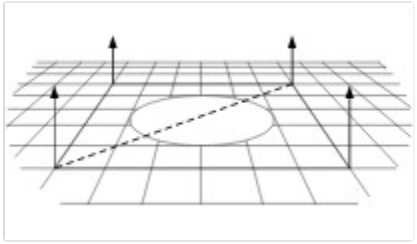


Throughout Direct3D, vertices describe position and orientation. Each vertex in a primitive is described by a vector that gives its position and a normal vector that gives its orientation, texture coordinates, and a color.

Shade Modes

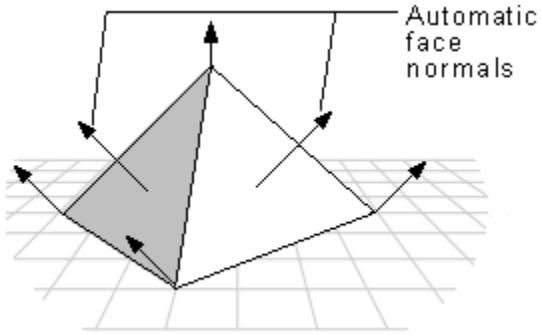
In the flat shade mode, the system duplicates the color of one vertex across all the other faces of the primitive. In the Gouraud and Phong shade modes, vertex normals are used to give a smooth look to a polygonal object. In Gouraud shading, the color and intensity of adjacent vertices is interpolated across the space that separates them. In Phong shading, which is not currently supported by Direct3D, the system calculates the appropriate shade value for each pixel on a face.

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.

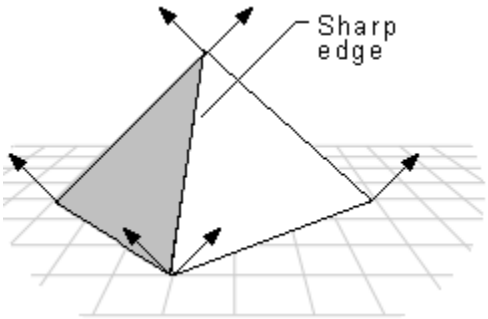


In this case, the Phong shade mode would calculate the value for each pixel and display the spotlight. The Gouraud shade mode, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

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 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. These 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 currently supported.

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 or "ramp" model (**D3DCOLOR_MONO**), the system uses only the blue component of the vertex color.

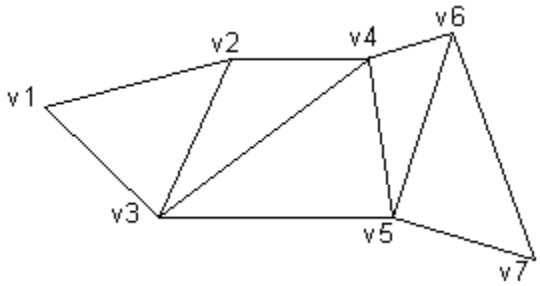
For example, if the red component of the color of vertex 1 were 0.8 and the red component of vertex 2 were 0.4, in the Gouraud shade mode and RGB color model the system would use interpolation to assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

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.

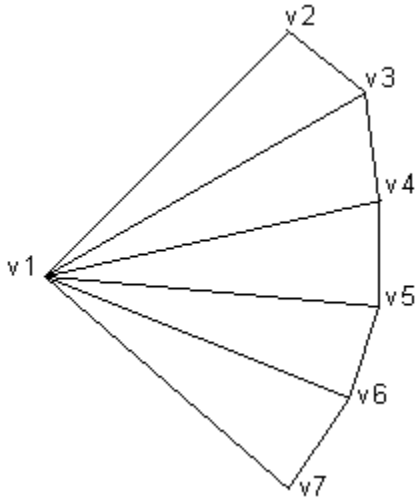
Triangle Strips and Fans

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 v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the vertices of the second and fourth triangles 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 v1, v2, and v3 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v1, v4, and v5 to draw the third triangle, and so on.

You can use the **wFlags** member of the **D3DTRIANGLE** structure to specify the flags that build triangle strips and fans.

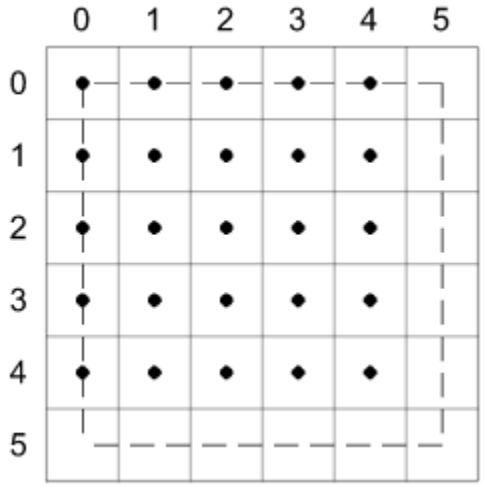
Triangle Rasterization Rules

Often the points specified for vertices do not precisely match the pixels on the screen. When this happens, Direct3D applies triangle rasterization rules to decide which pixels apply to a given triangle.

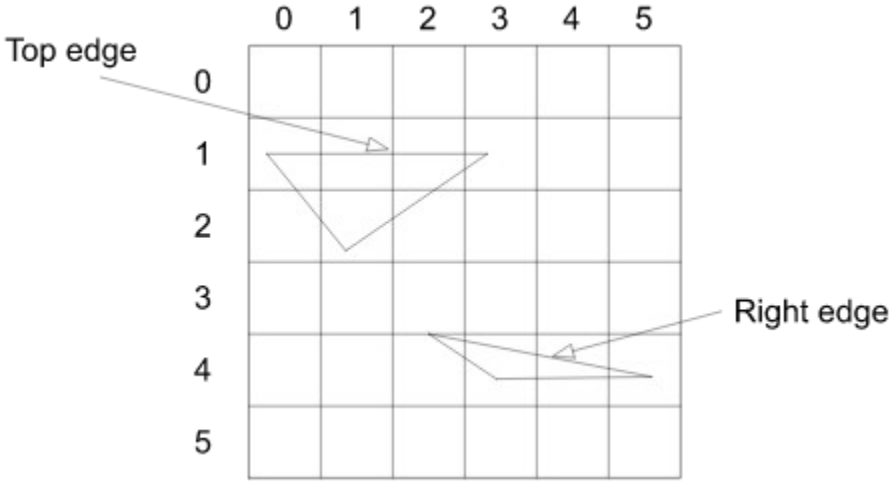
Direct3D uses a top-left filling convention for filling geometry. This is the same convention that is used for rectangles in GDI, the Windows NT polygon rasterizer, and OpenGL. Also, in Direct3D the center of the pixel is the point at which decisions are made; if the center is inside a triangle, the pixel is part of the triangle. Pixel centers are at integer coordinates.

This description of triangle-rasterization rules used by Direct3D does not necessarily apply to all available hardware. Your testing may uncover minor variations in the implementation of these rules. In the future, nearly all manufacturers will implement these rules as they are described in this section.

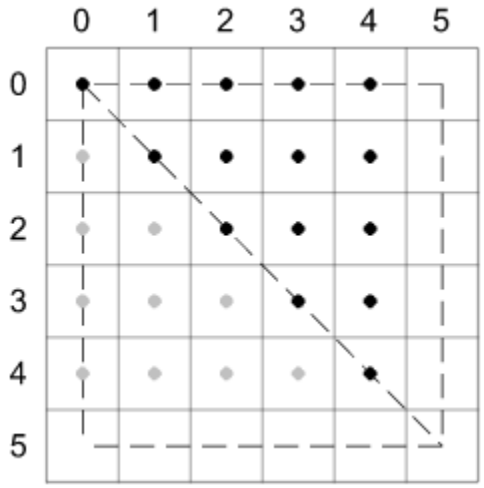
The following illustration shows a rectangle whose upper-left corner is at [0, 0] and whose lower-right corner is at [5, 5]. This rectangle fills 25 pixels, just as you would expect. The width of the rectangle is defined as right-left. The height is defined as bottom-top.



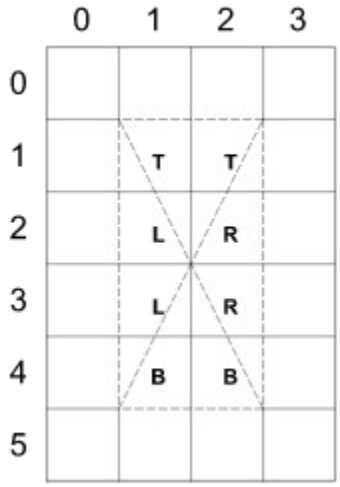
In the top-left filling convention, the word "top" refers to horizontal spans, and the word "left" refers to pixels in spans. An edge cannot be a top edge unless it is horizontal—in the general case, most triangles will have only left and right edges.



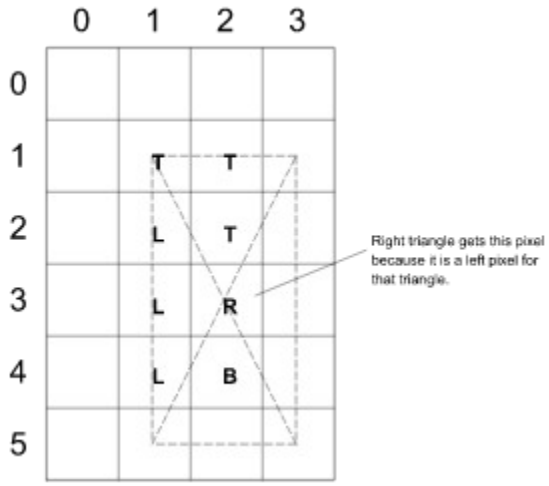
The top-left filling convention determines the action taken by Direct3D when a triangle passes through the center of a pixel. The following illustration shows two triangles, one at [0, 0], [5, 0], and [5, 5], and the other at [0, 5], [0, 0], and [5, 5]. The first triangle in this case gets 15 pixels, whereas the second gets only 10, because the shared edge is the left edge of the first triangle.



Suppose that a rectangle were defined with its upper-left corner at [0.5, 0.5] and its lower-right corner at [2.5, 4.5]. The center point of this rectangle would be at [1.5, 2.5]. When this rectangle was tessellated, the center of each pixel would be unambiguously inside each of the four triangles, and the top-left filling convention would not be needed.

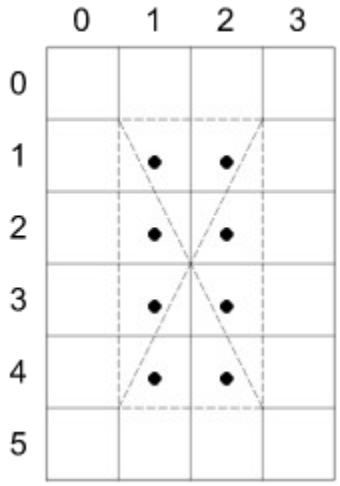


If a rectangle with the same dimensions were moved slightly, so that its upper-left corner were at [1.0, 1.0], its lower-right corner at [3.0, 5.0], and its center point at [2.0, 3.0], the top-left filling convention would be required. Most of the pixels in this new rectangle would straddle the border between two or more triangles.

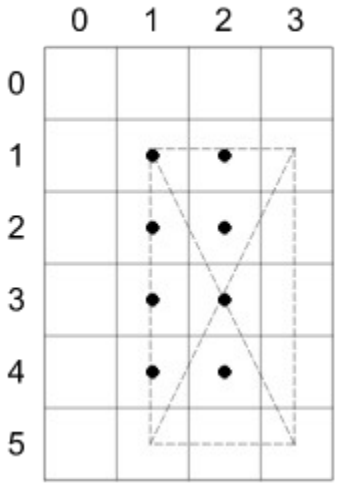


Notice that for both rectangles, the same pixels are affected.

(0.5, 0.5)-(2.5, 4.5)



(1.0, 1.0)-(3.0, 5.0)



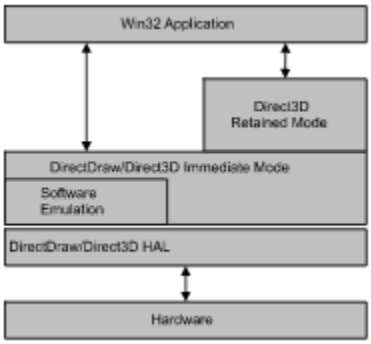
Direct3D Immediate-Mode Architecture

This section provides high-level information about the organization of the Direct3D Immediate Mode documentation. Information is divided into the following groups:

- Architectural Overview
- Immediate Mode Object Types
- Immediate Mode COM Interfaces
- The DrawPrimitive Methods and Execute Buffers

Architectural Overview

Direct3D applications communicate with graphics hardware in a similar fashion, whether they use Retained Mode or Immediate Mode. They may or may not take advantage of software emulation before interacting with the HAL. Since Direct3D is an interface to a DirectDraw® object, the HAL is referred to as the DirectDraw/Direct3D HAL.



Direct3D is tightly integrated with the DirectDraw component of DirectX® DirectDraw surfaces are used as rendering targets (front and back surfaces) and as z-buffers. The Direct3D COM interface is actually an interface to a DirectDraw object.

Immediate Mode Object Types

Direct3D Immediate Mode is made up of a series of objects. You work with these objects to manipulate your virtual world and build a Direct3D application.

DirectDraw Object

A DirectDraw object provides the functionality of Direct3D; **IDirect3D** and **IDirect3D2** are interfaces to a DirectDraw object. Since a DirectDraw object represents the display device, and the display device implements many of the most important features of Direct3D, it makes sense that the abilities of Direct3D are incorporated into DirectDraw. You create a DirectDraw object by calling the **DirectDrawCreate** function. For more information, see IDirect3D2 Interface.

DirectDrawSurface Object

A DirectDrawSurface object that was created as a texture map contains the bitmap(s) that your Direct3D application will use as textures. You create an **IDirect3DTexture2** interface by calling the **IDirectDrawSurface3::QueryInterface** method.

For more information, see Textures.

Direct3DDevice Object

A Direct3DDevice object encapsulates and stores the rendering state for an Immediate Mode application; it can be thought of as a rendering target for Direct3D. Prior to DirectX 5, Direct3D devices were interfaces to DirectDrawSurface objects. DirectX 5 introduced a new device-object model, in which a Direct3DDevice object is entirely separate from DirectDraw surfaces. This new object supports the **IDirect3DDevice2** interface.

You can call the **IDirect3D2::CreateDevice** method to create a Direct3DDevice object and retrieve an **IDirect3DDevice2** interface. (Notice that you do not call **QueryInterface** to retrieve **IDirect3DDevice2**!) If necessary, you can retrieve an **IDirect3DDevice** interface by calling the **IDirect3DDevice2::QueryInterface** method.

For more information, see The DrawPrimitive Methods and Execute Buffers and Devices.

Direct3DMaterial Object

A Direct3DMaterial object describes the illumination properties of a visible element in a three-dimensional scene, including how it handles light and whether it uses a texture. You can create a Direct3DMaterial object by calling the **IDirect3D2::CreateMaterial** method. You can use the **IDirect3DMaterial2** interface to get and set materials and to retrieve material handles.

For more information, see Materials.

Direct3DViewport Object

A Direct3DViewport object defines the rectangle into which a three-dimensional scene is projected. You can create an **IDirect3DViewport2** interface by calling the **IDirect3D2::CreateViewport** method. For more information, see Viewports and Transformations.

Direct3DLight Object

A Direct3DLight object describes the characteristics of a light in your application. You can use the **IDirect3DLight** interface to get and set lights. You can create an **IDirect3DLight** interface by calling the **IDirect3D2::CreateLight** method. For more information, see Lights.

Direct3DExecuteBuffer Object

A Direct3DExecuteBuffer object is a buffer full of vertices and instructions about how to handle them. Prior to DirectX 5, Immediate-Mode programming was done exclusively by using Direct3DExecuteBuffer objects. The introduction of the DrawPrimitive methods in DirectX 5, however, has made it unnecessary for most applications to work with execute buffers.

For more information about execute buffers, see Execute Buffers.

Immediate Mode COM Interfaces

The Direct3D Immediate Mode API consists of the following COM interfaces:

IDirect3D2	Root interface, used to obtain other interfaces
IDirect3DDevice	3D Device for execute-buffer based programming
IDirect3DDevice2	3D Device for DrawPrimitive-based programming
IDirect3DTexture2	Texture-map interface
IDirect3DMaterial2	Surface-material interface
IDirect3DViewport2	Interface to define the screen space viewport's characteristics.
IDirect3DLight	Interface used to work with lights
IDirect3DExecuteBuffer	Interface for working with execute buffers

For backward compatibility with previous versions of DirectX, the following interfaces are also provided. For more information about backward compatibility, see Compatibility with DirectX 3.

IDirect3D
IDirect3DTexture
IDirect3DMaterial
IDirect3DViewport

The DrawPrimitive Methods and Execute Buffers

DirectX 5 introduced a radically new way to use Direct3D Immediate Mode. Previously, you had to fill and execute the execute buffers to accomplish any task. Now, you can use the DrawPrimitive methods, which allow you to draw primitives directly.

The **IDirect3DDevice** interface supports execute buffers. The **IDirect3DDevice2** interface supports the DrawPrimitive methods. Despite the names of these interfaces, **IDirect3DDevice2** is not a COM iteration of the **IDirect3DDevice** interface. Although there is some overlap in the functionality of the interfaces, they are separate implementations. This means that you cannot call **IDirect3DDevice::QueryInterface** to retrieve an **IDirect3DDevice2** interface. You must call the **IDirect3D2::CreateDevice** method, instead.

For more information about the DrawPrimitive methods, see The DrawPrimitive Methods. For more information about working with execute buffers, see Using Execute Buffers. For more information about device objects, see Objects and Interfaces.

Direct3D Immediate-Mode Essentials

Direct3D Immediate Mode consists of a relatively small number of API elements that create objects, fill them with data, and link them together. The API is based on the COM model. The Immediate Mode API are a very thin layer over the Direct3D drivers.

This section provides technical information about the components Direct3D Immediate Mode. Information is divided into the following groups.

Immediate-Mode Changes for DirectX 5

This section discusses some of the important changes in the implementation of Direct3D Immediate Mode for DirectX 5. Information is divided into the following groups.

- Compatibility with DirectX 3
- Moving DirectX 3 Applications to DirectX 5

Compatibility with DirectX 3

Direct3D Immediate Mode has undergone many changes between DirectX 3 and DirectX 5. One of the benefits of the COM model is that it allows additions and modifications to sets of API elements without breaking existing applications. All of the DirectX 3 interfaces are supported in DirectX 5. The DirectX 3 interfaces also work in a similar way (except for a few bug fixes).

The new functionality in DirectX 5 required the addition of many new interfaces. The most important changes in DirectX 5 have been the addition of the DrawPrimitive methods and the new device object model. For more information about the DrawPrimitive methods, see The DrawPrimitive Methods. For more information about the new device object model, see Objects and Interfaces.

Moving DirectX 3 Applications to DirectX 5

Although DirectX 3 applications will work unchanged in DirectX 5, DirectX 3 applications should be modified if the new DirectX 5 features are to be used. The main change required is use the new Direct3D device model described in Objects and Interfaces. Retrieve an **IDirect3D2** interface from DirectDraw instead of **IDirect3D** and then use the **IDirect3D2::CreateDevice** method to create the device object. You can continue to use the **IDirect3DDevice** interface just as you did previously by calling the **IDirect3DDevice2::QueryInterface** method to retrieve the **IDirect3DDevice** interface. The only difference is that if you previously called **IDirect3DDevice::QueryInterface** to retrieve an **IDirectDrawSurface** interface, you should instead call **IDirect3DDevice2::GetRenderTarget**.

The DrawPrimitive Methods

This section discusses the DrawPrimitive methods, an innovation in DirectX 5 that both simplifies Immediate Mode programming and adds new flexibility. Information is divided into the following groups.

API Extensions for DrawPrimitive

There is a way to tap into the power of Immediate Mode programming without explicitly using execute buffers. The heart of this system is the **IDirect3DDevice2::DrawPrimitive** method (and its companion, **IDirect3DDevice2::DrawIndexedPrimitive**).

The **IDirect3D** and **IDirect3DDevice** interfaces have been extended to support the ability to draw primitives. These extended versions are called the **IDirect3D2** and **IDirect3DDevice2** interfaces.

To use **IDirect3DDevice2**, retrieve a pointer to the interface by calling the **IDirect3D2::CreateDevice** method. If you need to use some of the methods in **IDirect3DDevice** that are not supported in **IDirect3DDevice2**, you can call **IDirect3DDevice2::QueryInterface** to retrieve a pointer to an **IDirect3DDevice** interface.

The **IDirect3DViewport** interface has also been extended. The new interface, **IDirect3DViewport2**, introduces a closer correspondence between the dimensions of the clipping volume and the viewport than was true for the **IDirect3DViewport** interface.

Architecture of DrawPrimitive Capabilities

The methods provided by the **IDirect3D2** and **IDirect3DDevice2** interfaces enable a user to avoid the execute buffer model. Avoiding the execute buffer model can be useful for two reasons:

1. A new Direct3D developer wants to get up and running quickly.
2. Certain classes of applications (for example, BSP-style games with graphics engines that produce transformed, lit and clipped triangles) do not lend themselves easily to porting to the execute-buffer model. The execute-buffer model does not inherently impose restrictions, but it can be difficult to use.

The **IDirect3DDevice2** interface can be created with the **IDirect3D2::CreateDevice** method. This method takes the DirectDraw surface to render into as a parameter, enabling applications to avoid querying the device interface off of the DirectDraw surface.

A "primitive" in the DrawPrimitive API can be one of the following constructs:

- Point list
- Line list
- Line strip
- Triangle list
- Triangle strip
- Triangle fan

The **IDirect3DDevice2::DrawPrimitive** and **IDirect3DDevice2::DrawIndexedPrimitive** methods draw a primitive in a single call. When possible, these functions call into the driver directly to draw the primitive.

Alternatively, the application can specify the vertices one at a time. To draw a primitive by specifying the vertices individually, the application calls **IDirect3DDevice2::Begin** and specifies the primitive and vertex type, **IDirect3DDevice2::Vertex** to specify each vertex, and **IDirect3DDevice2::End** to finish drawing the primitive. Similarly, to draw an indexed primitive by specifying the indices individually, the application calls **IDirect3DDevice2::BeginIndexed** and specifies the primitive and vertex type, **IDirect3DDevice2::Index** to specify each index, and **IDirect3DDevice2::End** to finish drawing the primitive.

IDirect3DDevice2::Vertex is the only valid method between calls to **IDirect3DDevice2::Begin** and **IDirect3DDevice2::End**.

IDirect3DDevice2::Index is the only valid method between calls to **IDirect3DDevice2::BeginIndexed** and **IDirect3DDevice2::End**.

Note The DrawPrimitive methods are designed to enable asynchronous operation. Unless you specify D3DDP_WAIT when you call **IDirect3DDevice2::DrawPrimitive** or **IDirect3DDevice2::DrawIndexedPrimitive**, the method will fail if the 3-D hardware cannot currently accept the command, returning DDERR_WASSTILLDRAWING . This behavior is modeled after the DirectDraw **IDirectDrawSurface3::Blt** operation, where DDBLT_WAIT specifies that the call should return after the command has actually been queued up for execution by the accelerator.

Using Both DrawPrimitive and Execute Buffers

DirectX 5 applications can use both styles of programming in the same application by using the two interfaces to the device object. The application should retrieve an **IDirect3D2** interface by calling the **QueryInterface** method on the DirectDraw object and then use the **IDirect3D2::CreateDevice** method to create a device object. The application should keep the **IDirect3DDevice2** interface thus obtained and also call **IDirect3DDevice2::QueryInterface** to retrieve an **IDirect3DDevice** interface.

It is recommended that you use **IDirect3DMaterial2** and **IDirect3DTexture2** interfaces (created using **IDirect3D2**). You can get handles from these objects using the **IDirect3DDevice2** interface as an argument to the **IDirect3DMaterial2::GetHandle** and **IDirect3DTexture2::GetHandle** methods. You can use these handles both in **IDirect3DDevice2** methods and in execute buffers rendered through **IDirect3DDevice**. This is because **IDirect3DDevice2** and **IDirect3DDevice** are two interfaces to the same underlying object.

Similarly, you can create viewport objects using **IDirect3D2** and use the new **IDirect3DViewport2** interface. These viewport objects can be added to the device using **IDirect3DDevice2::AddViewport**. Because the **IDirect3DViewport2** interface inherits from **IDirect3DViewport**, you can pass it to **IDirect3DDevice** methods that expect the **IDirect3DViewport** interface.

A Simple DrawPrimitive Example

For a complete working example of an application that uses the DrawPrimitive methods, see the files in the Flip3D directory in the samples that ship with the DirectX 5 SDK.

```
/*
 * Constants
 */
#define NUM_VERTICES 3
#define NUM_TRIANGLES 1
D3DTLVERTEX src_v[NUM_VERTICES];
WORD src_t[NUM_TRIANGLES * 3];
DWORD hTex;
D3DSTATEVALUE hMat;

/*
 * A routine that assumes that the above data is initialized
 */

BOOL RenderScene(LPDIRECT3DDEVICE2 lpDev, LPDIRECT3DVIEWPORT lpView,
    LPD3DRECT lpExtent)
{
    if (IDirect3DDevice2_BeginScene(lpDev) != D3D_OK)
        return FALSE;
    if (IDirect3DDevice2_SetLightState(lpDev, D3DLIGHTSTATE_MATERIAL,
            hMat) != D3D_OK)
        return FALSE;
    if (IDirect3DDevice2_SetRenderState(lpDev,
            D3DRENDERSTATE_TEXTUREHANDLE, hTex) != D3D_OK)
        return FALSE;
    if (IDirect3DDevice2_DrawIndexedPrimitive(lpDev,
            DPT_TRIANGLELIST, DVT_TLVERTEX,
            (LPVOID)src_v, NUM_VERTICES,(LPWORD)src_t, NUM_TRIANGLES*3)
             != D3D_OK)
        return FALSE;
    if (IDirect3DDevice2_EndScene(lpDev) != D3D_OK)
        return FALSE;

    return TRUE;
}
```

GUIDs

Direct3D uses globally unique identifiers, or GUIDs, to identify parts of the interface. When you use the **QueryInterface** method to determine whether an object supports an interface, you identify the interface you're interested in by using its GUID.

To use GUIDs successfully in your application, you must either define INITGUID prior to all other include and define statements, or you must link to the DXGUID.LIB library. You should define INITGUID in only one of your source modules.

Note that you use GUIDs differently depending on whether your application is written in C or C++. In C, you pass a pointer to the GUID (&IID_IDirect3D, for example), but in C++, you pass a reference to it (simply IID_IDirect3D).

IDirect3D2 Interface

The **IDirect3D2** interface is the starting point for creating other Direct3D Immediate Mode interfaces. It is implemented by the DirectDraw object and can be obtained by calling the **IDirectDraw2::QueryInterface** method.

IDirect3D2 lets you find and enumerate the types of Direct3D devices supported by a particular DirectDraw object. It also has methods to create other Direct3D Immediate Mode objects, such as viewports, materials and lights.

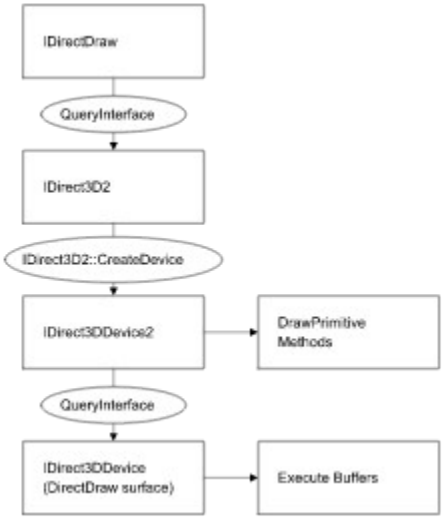
The most important difference between **IDirect3D2** and its predecessor, **IDirect3D**, is that **IDirect3D2** implements an **IDirect3D2::CreateDevice** method. This method creates a Direct3D device that supports the DrawPrimitive methods. For more information about the devices created by the **CreateDevice** method, see Devices.

Devices

A Direct3D device can be thought of as a rendering target for Direct3D. It encapsulates and stores the rendering state.

The Direct3D Immediate Mode device object supports two interfaces, **IDirect3DDevice** and **IDirect3DDevice2**. These two interfaces abstract two styles of programming in Direct3D Immediate Mode. **IDirect3DDevice** allows programming using execute buffers. **IDirect3DDevice2** is a more immediate interface that allows you to draw primitives directly. The interfaces share a few common methods that are useful in either programming style. These have been replicated in both the interfaces to reduce the need to call the **QueryInterface** method between each interface.

You can call the **IDirect3D2::CreateDevice** method to create a Direct3D device object. This method retrieves an **IDirect3DDevice2** interface. The object, however supports both device interfaces; you can retrieve an **IDirect3DDevice** interface by calling the **IDirect3DDevice2::QueryInterface** method.



When you create an **IDirect3DDevice2** interface, the Direct3D device object is a separate object from a DirectDraw surface object. The device object uses a DirectDraw surface as a rendering target. This behavior is different from **IDirect3DDevice**, in which Direct3D devices are simply interfaces to DirectDraw surfaces. Keeping devices as separate objects with independent lifetimes from DirectDraw surfaces allows a single 3-D device object to use different DirectDraw surfaces as render targets at different times (for information about this, see **IDirect3DDevice2::SetRenderTarget**). For more information on backward compatibility and the differences between DirectX 3 and DirectX 5, see Compatibility with DirectX 3.

Viewports and Transformations

Direct3D uses three transformations: the view transform, the world transform, and the projection transform. Understanding how these are applied is critical to getting the results that you expect.

This section contains the following topics related to viewports and transformations:

The Transformation Pipeline

Models are normally created centered around a natural local origin. For instance, it makes sense to have the origin of a chair model be at floor level and centered under the chair. This helps make it easier to place the model in the world. The coordinates that define the model are relative to the origin of the chair model, of course, and are known as model coordinates.

The world transform controls how geometry is transformed from model coordinates into world coordinates. This transform can include translations, rotations, and scalings. You would use the world transform to place your chair model in a room and scale it with respect to the other objects in the room. The world transform applies only to geometry − it does not apply to lights. For an example of working with world transforms, see World Transform.

The view transform controls the transition from world coordinates into "camera space." You can think about this transformation as controlling where the camera appears to be in the world. For an example of working with view transforms, see View Transform.

The projection transform changes the geometry from camera space into "clip space" and applies the perspective distortion. The term "clip space" refers to how the geometry is clipped to the view volume during this transform. For an example of working with projection transforms, see Projection Transform.

Finally, the geometry in clip space is transformed into pixel coordinates (screen space). This final transformation is controlled by the viewport settings.

Clipping and transforming vertices must take place in homogenous space (simply put, space in which the coordinate system includes a fourth element), but the final result for most applications needs to be non-homogenous 3-D coordinates defined in "screen space." This means that both the input vertices and the clipping volume must be translated into homogenous space to perform the clipping and then translated back into non-homogenous space to be displayed.

The world, view and projection matrices are multiplied in that order to produce the combined transformation matrix [M]. An input vertex [x y z] is considered to be a homogenous vertex [x y z 1]. This vertex is multiplied by the combined 4×4 transform matrix [M] to obtain the output vertex [x1 y1 z1 w]. Following this multiplication, all input vertices are in "post-perspective homogenous space." Now that the vertices have been transformed and changed into homogenous space, the same thing must happen to the clipping volume; it is transformed into the post-perspective homogenous space and the clipping is performed. These clipped vertices (all of which lie within the clip volume) are now transformed back into post-perspective non-homogenous space. As a final step, the points are scaled so that the clip volume maps to the screen space viewport specified by the **dwX**, **dwY**, **dwHeight**, **dwWidth** members of the **D3DVIEWPORT2** structure.

Setting Transformations

Transformations are represented by a 4×4 matrix and are applied using the **IDirect3DDevice2::SetTransform** method. For example, you could use code like this to set the view transform:

```
D3DMATRIX  view;

// fill in the view matrix…

if ((err=lpDev->SetTransform(D3DTRANSFORMSTATE_VIEW, &view)) != D3D_OK)
return err;
```

The render states for setting the projection and the world transformations are D3DTRANSFORMSTATE_PROJECTION and D3DTRANSFORMSTATE_WORLD, respectively.

Creating and Deleting Viewports

A viewport is the surface rectangle into which a three-dimensional scene is projected. A Direct3D viewport object is used to specify the following:

- The screen-space viewport to which the rendering will be confined.
- The post-transform clip volume, the contents of which will be mapped in to the viewport. (This is also known as the "window" in a "window-to-viewport transform," in standard computer-graphics terminology.)
- The background material and texture to which the viewport should be cleared.
- The background depth buffer used to initialize the *z*-buffer before rendering the scene.

The **IDirect3DViewport2** interface has two ways of specifying a viewport. The **D3DVIEWPORT2** structure is specified by the new methods in **IDirect3DViewport2**. This is similar to the **D3DVIEWPORT** structure except that it allows a better clip-volume definition. The new viewport structure is recommended for all DirectX 5 applications.

The first thing to do when creating a viewport is to create a **D3DVIEWPORT2** structure and a pointer to a viewport object:

```
LPDIRECT3DVIEWPORT2  lpD3DViewport;
D3DVIEWPORT2    viewData;
```

Next, fill in the viewport structure:

```
float  aspect = (float)width/height;    // aspect ratio of surface

memset(&viewData, 0, sizeof(D3DVIEWPORT2));
viewData.dwSize = sizeof(D3DVIEWPORT2);
viewData.dwX = 0;
viewData.dwY = 0;
viewData.dwWidth = width;
viewData.dwHeight = height;
viewData.dvClipX = -1.0f;
viewData.dvClipY = aspect;
viewData.dvClipWidth = 2.0f;
viewData.dvClipHeight = 2.0f * aspect;
viewData.dvMinZ = 0.0f;
viewData.dvMaxZ = 1.0f;
```

(You can find a discussion of setting the clipping volume in this structure in the reference material for **D3DVIEWPORT2**.)

After filling in this structure, call the **IDirect3D2::CreateViewport** method to create the viewport object. For this you need a valid **LPDIRECT3D2** pointer, shown in the following example as *lpD3D2*.

```
if ((err = lpD3D2->CreateViewport(&lpD3DViewport2, NULL)) != D3D_OK) {
  return err;
}
```

Now you can call the **IDirect3DDevice2::AddViewport** method to add the newly created viewport object to the device. For this you need a valid **LPDIRECT3DDEVICE2** pointer, shown in the following example as *lpD3DDevice2*.

```
if ((err = d3dapp->lpD3DDevice2->AddViewport(lpD3DViewport2)) != D3D_OK) {
  return err;
}
```

Finally, call the **IDirect3DViewport2::SetViewport2** method to associate the **D3DVIEWPORT2** structure whose values you have already filled out with the new viewport object.

```
if ((err = lpD3DViewport2->SetViewport2(&viewData)) != D3D_OK) {
return err;
}
```

At this point you have a working viewport. If you need to make changes to the viewport values, simply update the values in the **D3DVIEWPORT2** structure and call **IDirect3DViewport2::SetViewport2** again.

When you are ready to delete the viewport, first delete any lights and materials associated with it and then call the **IDirect3DViewport2::Release** method.

```
lpD3DViewport2->Release();
```

Matrices

A Direct3D matrix is a 4×4 homogenous matrix, as defined by a **D3DMATRIX** structure. You use Direct3D matrices to define world, view, and projection transformations. Although Direct3D matrices are not standard objects − they are not represented by a COM interface − you can create and set them just as you would any other Direct3D object.

The D3D_OVERLOADS implementation of the **D3DMATRIX** structure (**D3DMATRIX (D3D_OVERLOADS)**) implements a parentheses ("()") operator. This operator offers convenient access to values in the matrix for C++ programmers. Instead of having to refer to the structure members by name, C++ programmers can refer to them by row and column number, and simply index these numbers as needed. These indices are zero-based, so for example the element in the third row, second column would be M(2, 1). To use the D3D_OVERLOADS operators, you must define D3D_OVERLOADS before including D3dtypes.h.

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.

Matrices appear to you only as handles. These handles (defined by the **D3DMATRIXHANDLE** type) are used in execute buffers and in the **D3DOP_MATRIXLOAD** and **D3DOP_MATRIXMULTIPLY** opcodes.

World Transform

The world transform changes coordinates from model space to world space. This can include any combination of translations, rotations, and scalings. For a discussion of the mathematics of transformations, see 3-D Transformations.

You can create a translation using code like this. Notice that here (and in the other transformation samples) the D3D_OVERLOADS form of **D3DMATRIX** is being used.

```
D3DMATRIX Translate(const float dx, const float dy, const float dz)
{
    D3DMATRIX ret = IdentityMatrix();
    ret(3, 0) = dx;
    ret(3, 1) = dy;
    ret(3, 2) = dz;
    return ret;
}    // end of Translate()
```

You can create a rotation around an axis using code like this:

```
D3DMATRIX RotateX(const float rads)
{
    float    cosine, sine;

    cosine = cos(rads);
    sine = sin(rads);
    D3DMATRIX ret = IdentityMatrix();
    ret(1,1) = cosine;
    ret(2,2) = cosine;
    ret(1,2) = -sine;
    ret(2,1) = sine;
    return ret;
}   // end of RotateX()

D3DMATRIX RotateY(const float rads)
{
    float    cosine, sine;

    cosine = cos(rads);
    sine = sin(rads);
    D3DMATRIX ret = IdentityMatrix();
    ret(0,0) = cosine;
    ret(2,2) = cosine;
    ret(0,2) = sine;
    ret(2,0) = -sine;
    return ret;
}   // end of RotateY()

D3DMATRIX RotateZ(const float rads)
{
    float    cosine, sine;

    cosine = cos(rads);
    sine = sin(rads);
    D3DMATRIX ret = IdentityMatrix();
    ret(0,0) = cosine;
```

```
    ret(1,1) = cosine;
    ret(0,1) = -sine;
    ret(1,0) = sine;
    return ret;
}   // end of RotateZ()
```

You can create a scale transform using code like this:

```
D3DMATRIX Scale(const float size)
{
D3DMATRIX ret = IdentityMatrix();
    ret(0, 0) = size;
    ret(1, 1) = size;
    ret(2, 2) = size;
    return ret;
}   // end of Scale()
```

These basic transformations can be combined to create the final transform. Remember that when you combine them the results are not commutative – the order in which you multiply matrices is important.

View Transform

The view transform changes coordinates from world space to camera space. In effect, this transform moves the world around so that the right parts of it are in front of the camera, given a camera position and orientation.

The following *ViewMatrix* function creates a view matrix based on the camera location passed to it. It uses the **Normalize**, **CrossProduct**, and **DotProduct** D3D_OVERLOADS helper functions. The *RotateZ* function it uses is shown in the World Transform section.

```
D3DMATRIX
ViewMatrix(const D3DVECTOR from,      // camera location
           const D3DVECTOR at,        // camera look-at target
           const D3DVECTOR world_up,  // world's up, usually 0, 1, 0
           const float roll)          // clockwise roll around
                                      //    viewing direction,
                                      //    in radians
{
    D3DMATRIX  view = IdentityMatrix();  // shown below
    D3DVECTOR  up, right, view_dir;

    view_dir = Normalize(at - from);
    right = CrossProduct(world_up, view_dir);
    up = CrossProduct(view_dir, right);

    right = Normalize(right);
    up = Normalize(up);

    view(0, 0) = right.x;
    view(1, 0) = right.y;
    view(2, 0) = right.z;
    view(0, 1) = up.x;
    view(1, 1) = up.y;
    view(2, 1) = up.z;
    view(0, 2) = view_dir.x;
    view(1, 2) = view_dir.y;
    view(2, 2) = view_dir.z;

    view(3, 0) = -DotProduct(right, from);
    view(3, 1) = -DotProduct(up, from);
    view(3, 2) = -DotProduct(view_dir, from);

    if (roll != 0.0f) {
        // MatrixMult function shown below
        view = MatrixMult(RotateZ(-roll), view);
    }

    return view;
    }  // end of ViewMatrix()

D3DMATRIX
IdentityMatrix(void)       // initializes identity matrix
{
    D3DMATRIX ret;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
```

```
            ret(i, j) = (i==j) ? 1.0f : 0.0f;
    return ret;
}    // end of IdentityMatrix()

// Multiplies two matrices.
D3DMATRIX
MatrixMult(const D3DMATRIX a, const D3DMATRIX b)
{
    D3DMATRIX ret = ZeroMatrix(); // shown below

    for (int i=0; i<4; i++) {
        for (int j=0; j<4; j++) {
            for (int k=0; k<4; k++) {
                ret(i, j) += a(k, j) * b(i, k);
            }
        }
    }
    return ret;
}    // end of MatrixMult()

D3DMATRIX
ZeroMatrix(void)  // initializes matrix to zero
{
    D3DMATRIX ret;
    for (int i=0; i<4; i++)
        for (int j=0; j<4; j++)
            ret(i, j) = 0.0f;
    return ret;
}    // end of ZeroMatrix()
```

Projection Transform

You can think of the projection transform as controlling the camera's internals. The following *ProjectionMatrix* function takes three input parameters that set the near and far clipping planes and the field of view angle. The field of view should be less than pi.

```
D3DMATRIX
ProjectionMatrix(const float near_plane,     // distance to near clipping
plane
                 const float far_plane,      // distance to far clipping
plane
                 const float fov)            // field of view angle, in
radians
{
    float    c, s, Q;

    c = (float)cos(fov*0.5);
    s = (float)sin(fov*0.5);
    Q = s/(1.0f - near_plane/far_plane);

    D3DMATRIX ret = ZeroMatrix();
    ret(0, 0) = c;
    ret(1, 1) = c;
    ret(2, 2) = Q;
    ret(3, 2) = -Q*near_plane;
    ret(2, 3) = s;
    return ret;
}   // end of ProjectionMatrix()
```

The following matrix is the projection matrix used by Direct3D. In this formula, *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:

$$P = \begin{bmatrix} D/hF & 0 & 0 & 0 \\ 0 & D/hF & 0 & 0 \\ 0 & 0 & F/(F-D) & 1 \\ 0 & 0 & (-F*D)/(F-D) & 0 \end{bmatrix}$$

In Direct3D, the 3,4 element of the projection matrix (the numeral 1 here) cannot be a negative number.

Textures

A texture is a rectangular array of colored pixels. You can think of it as a source of *texels* (texture elements) for the rasterizer. Although the rectangular texture does not necessarily have to be square, 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-, 16-, 24-, and 32-bit textures. If you use the monochromatic (or ramp) color model, however, you can use only 8-bit textures.

This section describes Direct3D textures and the ways your applications can use them.

Surfaces, Devices, and Handles

You can use a DirectDraw surface as a texture map by calling the **IDirectDrawSurface::QueryInterface** method to retrieve an **IDirect3DTexture2** interface. You can use the **IDirect3DTexture2** interface to load textures, retrieve handles, and track changes to palettes.

IDirect3D2 and its DirectX 3 counterpart, **IDirect3DTexture**, can be associated with a 3-D device. A texture handle identifies this coupling of a texture map with a device. A texture can be associated with more than one device. When you call the **IDirect3DTexture2::GetHandle** method to associate a texture with a device, it is validated to ensure that the device can support the specified type of texture format and dimensions. The **GetHandle** method returns the texture handle if this validation succeeds. Texture handles can then be used as render states and in materials (for ramp mode) in either **IDirect3DDevice2** or **IDirect3DDevice**. Because a handle obtained by associating a texture object with a device object does not depend on which interfaces were used to obtain it, handles obtained using **IDirect3DTexture** or **IDirect3DTexture2** for a given device object can be used interchangeably.

The **IDirect3DTexture2** interface eliminates some unimplemented methods from the **IDirect3DTexture** interface.

The following example demonstrates how to create an **IDirect3DTexture2** interface and then how to load the texture by calling the **IDirect3DTexture2::GetHandle** and **IDirect3DTexture2::Load** methods. Note that the DirectDraw surface you query must have the **DDSCAPS_TEXTURE** capability to support a Direct3D texture.

```
lpDDS->QueryInterface(IID_IDirect3DTexture2,
    lpD3DTexture2);  // Address of a DIRECT3DTEXTURE object
lpD3DTexture2->GetHandle(
    lpD3DDevice,     // Address of a DIRECT3DDEVICE object
    lphTexture);     // Address of a D3DTEXTUREHANDLE
lpD3DTexture2->Load(
    lpD3DTexture);   // Address of a DIRECT3DTEXTURE object
```

To use a texture handle in an execute buffer, use the **D3DRENDERSTATE_TEXTUREHANDLE** render state (part of the **D3DRENDERSTATETYPE** enumerated type).

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.

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.

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_DESTBLEND** render states to specify how colors in the source and destination are combined. The combination options (called *blend factor*) 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. Each prefiltered image, or level, in the mipmap is a power of two smaller than the previous level. A high-resolution level is used for objects that are close to the viewer. Lower-resolution levels are used as the object moves farther away. Mipmapping is a computationally low-cost way of improving the quality of rendered textures.

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.

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

To create a surface representing a single level of a mipmap, specify the DDSCAPS_MIPMAP flag in the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method. Because all mipmaps are also textures, the DDSCAPS_TEXTURE flag must also be specified. It is possible to create each level manually and build the chain by using the **IDirectDrawSurface3::AddAttachedSurface** method. However, you can use the **IDirectDraw2::CreateSurface** method to build an entire mipmap chain in a single operation. In this case, the DDSCAPS_COMPLEX flag is also required.

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;
LPDIRECTDRAWSURFACE3 lpDDMipMap;
ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS | DDSD_MIPMAPCOUNT;
ddsd.dwMipMapCount = 5;
ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE |
    DDSCAPS_MIPMAP | DDSCAPS_COMPLEX;
ddsd.dwWidth = 256UL;
ddsd.dwHeight = 256UL;

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

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

A chain of mipmap surfaces is traversed by using the **IDirectDrawSurface3::GetAttachedSurface** method and specifying the DDSCAPS_MIPMAP and DDSCAPS_TEXTURE flags in the **DDSCAPS** structure. The following example traverses a mipmap chain from highest to lowest resolutions:

```
LPDIRECTDRAWSURFACE lpDDLevel, lpDDNextLevel;
DDSCAPS ddsCaps;
```

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

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

It is not possible to build such a surface arrangement with a single call to the **IDirectDraw2::CreateSurface** method. To construct a flipping mipmap, either build a complex mipmap chain and manually attach back buffers by using the **IDirectDrawSurface3::AddAttachedSurface** method, or create a sequence of flipping chains and build the mipmap by using **IDirectDrawSurface3::AddAttachedSurface**.

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

The **IDirectDrawSurface3::Flip** method will flip all the levels of a mipmap from the level supplied to the lowest level in the mipmap. A destination surface can also be provided, in which case all levels in the mipmap will flip to the back buffer in their flipping chain. This back buffer matches the supplied override. For example, if the third back buffer in the top-level flipping chain is supplied as the override, all levels in the mipmap will flip to the third back buffer.

The number of levels in a mipmap chain is stored explicitly. When an application obtains the surface description of a mipmap (by calling the **IDirectDrawSurface3::Lock** or **IDirectDrawSurface3::GetSurfaceDesc** method), the **dwMipMapCount** member of the **DDSURFACEDESC** structure will contain the number of levels in the mipmap, including the top level. For levels other than the top level in the mipmap, the **dwMipMapCount** member specifies the number of levels from that mipmap to the smallest mipmap in the chain.

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 the DirectDraw 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 support for color keys, see Color Keying in the DirectDraw documentation.

Lights

Surfaces are illuminated in your Direct3D application by the lights you create and position. You can use the **IDirect3DLight** interface to get and set lights. You can create an **IDirect3DLight** interface by calling the **IDirect3D2::CreateLight** method.

Lighting is only one of the variables controlling the final appearance of a visible element in a scene. The properties of a surface (including how it reflects light) are determined by materials, which are discussed in Materials. The shading of a surface defines how color is interpreted across a triangle; this is determined by the D3DRENDERSTATE_SHADEMODE render state, in the **D3DRENDERSTATETYPE** enumerated type. Finally, any texture that has been applied to a visible element also interacts with the lighting to change the object's appearance.

The simplest light type is an ambient light. An ambient light illuminates everything in the scene, regardless of the orientation, position, and surface characteristics of the objects in the scene. Because an 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.

The color and intensity of the current ambient light are states of the lighting module you can set. You can change the ambient light by using the **D3DLIGHTSTATE_AMBIENT** member of the **D3DLIGHTSTATETYPE** enumerated type.

Direct3D supports four specialized light types in addition to ambient lights. These are defined by the **D3DLIGHTTYPE** enumerated type. You can specify these light types and their capabilities by calling the **IDirect3DLight::SetLight** method and modifying the values in the **D3DLIGHT2** structure.

Point	A light source that radiates equally in all directions from its origin. Point light sources require the system to calculate a new lighting vector for every facet or normal they illuminate, and so are computationally more expensive than a parallel point light source. They produce a more faithful lighting effect than parallel point light sources, however.
Spotlight	A light source that 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. You can specify the angles of each of these two sections by modifying members of the **D3DLIGHT2** structure.
Directional	A light source that 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. Directional light has orientation but no position. It 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 light source that illuminates objects with

parallel light, but the orientation of the light is taken from the position of the light source. 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.

You use the **D3DCOLORVALUE** structure to specify the color of your lights. For more information, see Colored Lights in the Colors and Fog section.

Your application can use as many lights as the device supports. To find out how many lights a device supports, call the **IDirect3DDevice2::GetCaps** method and examine the **D3DLIGHTINGCAPS** structure.

Lighting is computationally intensive. By being careful about how you set up your lighting, you can achieve significant performance gains in your application. For detailed lighting performance information, see Lighting Tips in the Performance Optimization section.

To use a light, you first need to create a **D3DLIGHT2** structure and a pointer to a light object.

```
D3DLIGHT2        light;        // Structure defining the light
LPDIRECT3DLIGHT  lpD3DLight;   // Object pointer for the light
```

When you have done this, you need to fill in the **D3DLIGHT2** structure. The following example defines a white point light whose range is set to 10.0.

```
memset(&light, 0, sizeof(D3DLIGHT2));    // clear memory

light.dwSize = sizeof(D3DLIGHT2);        // required
light.dltType = D3DLIGHT_POINT;
light.dvPosition.x = 0.0f;               // set position
light.dvPosition.y = 10.0f;
light.dvPosition.z = 0.0f;
light.dcvColor.r = 1.0f;                 // set color to white
light.dcvColor.g = 1.0f;
light.dcvColor.b = 1.0f;
light.dvAttenuation0 = 0.0;              // set linear attenuation
light.dvAttenuation1 = 1.0;
light.dvAttenuation2 = 0.0;
light.dvRange = 10.0f;                   // set maximum range
light.dwFlags = D3DLIGHT_ACTIVE;         // enable light
```

The next step is to call the **IDirect3D2::CreateLight** method to create the light object. For this you need a valid **LPDIRECT3D2** interface pointer, *lpD3D2*.

```
if ((err = lpD3D2->CreateLight(&lpD3DLight, NULL) != D3D_OK)
return err;
```

When you have created the light object, you use the **D3DLIGHT2** structure you have already filled in to set its properties. Calling the **IDirect3DLight::SetLight** method associates the **D3DLIGHT2** structure with the light object you just created.

```
if ((err = lpD3DLight->SetLight((D3DLIGHT *)&light)) != D3D_OK)
return err;
```

Finally, you should call the **IDirect3DViewport2::AddLight** method to add this light to your current

viewport. (Each light source is bound to a single viewport.)

```
if ((err = lpView->AddLight(lpD3DLight)) != D3D_OK)
return err;
```

At this point you have a new light working with the current viewport. If you need to make changes to the light's values, simply update the **D3DLIGHT2** structure and call the **IDirect3DLight::SetLight** method again.

After you are finished working with the light, you should call the **IDirect3DViewport2::DeleteLight** method to remove the light from the viewport, and then call the **IDirect3DLight::Release** method.

```
if (lpView) {
lpView->DeleteLight(lpD3DLight);
}
RELEASE(lpD3DLight);
```

Materials

A material describes the illumination properties of a surface, including how it handles light and whether it uses a texture. You can also use a material to define the background for a viewport. You can create a material object by calling the **IDirect3D2::CreateMaterial** method. You can use the **IDirect3DMaterial2** interface to get and set materials and to retrieve material handles.

For the ramp-mode software device, the material object keeps track of the texture map used in conjunction with the material. This enables the pre-calculation of the material palettes. When you use textures in ramp mode, you must set the **D3DLIGHTSTATE_MATERIAL** member of the **D3DLIGHTSTATETYPE** enumerated type. Once the material properties are set, a material can be associated with a device. As with textures, a material handle identifies this association of a material and a device. A material can be associated with more than one device. You retrieve a material handle by calling the **IDirect3DMaterial2::GetHandle** method.

The current material is a state variable in a device as part of lighting related states. Handles obtained using **IDirect3DMaterial** or **IDirect3DMaterial2** for a given device object can be used interchangeably.

IDirect3DMaterial2 interface eliminates some unimplemented methods from the **IDirect3DMaterial** interface.

You can define the light-handling properties of a material in four ways:

Ambient	Specifies the color of the ambient light as reflected by the material.
Diffuse	Specifies the color of the diffuse light as reflected by the material. Diffuse light is light produced by one of the light sources described by the **D3DLIGHTTYPE** enumerated type (that is, any light except ambient light).
Specular	Specifies the color of reflected highlights as produced by the material.
Emissive	Specifies the color of the light that is emitted by the material.

These light-handling properties and other properties of the material, including the texture handle, are described by the **D3DMATERIAL** structure. You can use the **D3DLIGHTSTATE_MATERIAL** member of the **D3DLIGHTSTATETYPE** enumerated type to identify a material.

You can create an **IDirect3DMaterial2** interface by calling the **IDirect3D2::CreateMaterial** method. The following example demonstrates how to create an **IDirect3DMaterial2** interface. Then it demonstrates how to set the material and retrieve its handle by calling the **IDirect3DMaterial2::SetMaterial** and **IDirect3DMaterial2::GetHandle** methods.

```
lpDirect3D2->CreateMaterial(
    lplpDirect3DMaterial2,  // Address of a new material
    pUnkOuter);             // NULL
lpDirect3DMaterial2->SetMaterial(
    lpD3DMat);              // Address of a D3DMATERIAL structure
lpDirect3DMaterial2->GetHandle(
    lpD3DDevice2,           // Address of a DIRECT3DDEVICE object
    lpD3DMat);              // Address of a D3DMATERIAL structure
```

Colors and Fog

Colors in Direct3D are properties of vertices, textures, materials, faces, lights, and, of course, palettes.

This section describes the Direct3D palette and specular color value capabilities.

- Colored Lights
- Palette Entries
- Fog

Colored Lights

The **dcvColor** member of the **D3DLIGHT2** structure specifies a **D3DCOLORVALUE** structure. The colors defined by this structure are RGBA values that generally range from zero to one, with zero being black. Although you will usually want the light color to fall within this range, you can use values outside the range for special effects. For example, you could create a strong light that washes out a scene by setting the color to large values. You could also set the color to negative values to create a dark light, which actually removes light from a scene. Dark lights are useful for forcing dramatic shadows in scenes and other special effects.

When you use the ramp (monochromatic) lighting mode, the ambient light is built into the ramp, so you can't make your scene any darker than the current ambient light level. Also, remember that colored lights in RGB mode are converted into a gray-scale shade in ramp mode; a red light that looks good in RGB mode will be a dim white light in ramp mode.

Palette Entries

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. 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 (ramp) lighting mode, fog is implemented through the light states (that is, the **D3DLIGHTSTATETYPE** enumerated type). In the RGB lighting mode, or when you are working with a HAL, you implement fog by using the **D3DRENDERSTATE_FOGTABLESTART** and **D3DRENDERSTATE_FOGTABLEEND** values in the **D3DRENDERSTATETYPE** enumerated type.

There are three fog modes: linear, exponential, and exponential squared. Only the linear fog mode is currently supported.

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:

$$f = \frac{1}{2.71828^{(0.5 \times 0.8)}} = \frac{1}{1.4918} = 0.6703$$

Antialiasing

Antialiasing is a technique you can use to reduce the appearance of "jaggies" – the stair-step pixels used to draw any line that isn't exactly horizontal or vertical. In three-dimensional scenes, this artifact is most noticeable on the boundaries between polygons of different colors.

Direct3D supports two antialiasing techniques: edge antialiasing and general antialiasing. Which technique is best for your application depends on your requirements for performance and visual fidelity.

- Edge antialiasing
- General antialiasing

Edge Antialiasing

You can apply edge antialiasing to edges in a scene in your application. In edge antialiasing, you specify the edges in your scene that you want the system to antialias, and the system redraws those edges, averaging the values of neighboring pixels. Although this is not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

After drawing your scene, use the D3DPRASTERCAPS_ANTIALIASEDGES flag in the **D3DPRIMCAPS** structure to find out whether the current hardware supports edge antialiasing. If it does, set the D3DRENDERSTATE_EDGEANTIALIAS flag to TRUE. Now you can redraw the edges in the scene, using **IDirect3DDevice2::DrawPrimitive** and either the **D3DPT_LINESTRIP** or **D3DPT_LINELIST** primitive type.

Redrawing every edge in your scene will work without introducing major artifacts, but it can be computationally expensive. The most important edges to redraw are those between areas of very different color (for example, silhouette edges) or boundaries between very different materials.

When you have finished antialiasing, set D3DRENDERSTATE_EDGEANTIALIAS to FALSE.

General Antialiasing

Direct3D applies general antialiasing whenever each polygon or line is rendered – no separate pass is required.

On some hardware, general antialiasing can be applied only when the application has rendered the polygons sorted from back to front. To find out whether this is true of the current hardware, you can check the D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT flag (which means that the application must sort the polygons) and the D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT flag (which means that the application need not sort the polygons). These flags are part of the **dwRasterCaps** member of the **D3DPRIMCAPS** structure.

After finding out whether or not you need to sort the polygons, set the render state **D3DRENDERSTATE_ANTIALIAS** to D3DANTIALIAS_SORTDEPENDENT or D3DANTIALIAS_SORTINDEPENDENT and draw the scene.

When you no longer need general antialiasing, disable it by setting **D3DRENDERSTATE_ANTIALIAS** to D3DANTIALIAS_NONE.

Direct3D Integration with DirectDraw

This section contains information about the relationship between DirectDraw and Direct3D objects and interfaces, and about the DirectDraw 3-D-surface capabilities. The following topics are discussed:

- Objects and Interfaces
- Texture Maps
- Z-Buffers
- RGBZ Support

Objects and Interfaces

DirectDraw presents programmers with a single, unified object that encapsulates both the DirectDraw and Direct3D states. When you create a DirectDraw object and then use the **IDirectDraw2::QueryInterface** method to obtain an IDirect3D2 interface, the reference count of the DirectDraw object is 2.

The important implication of this is that the lifetime of the Direct3D driver state is the same as that of the DirectDraw object. Releasing the Direct3D interface does not destroy the Direct3D driver state. That state is not destroyed until all references to that object—whether they are DirectDraw or Direct3D references—have been released. Therefore, if you release a Direct3D interface while holding a reference to a DirectDraw driver interface, and then query the Direct3D interface again, the Direct3D state will be preserved.

In DirectX 2 and DirectX 3, a Direct3D device was aggregated off a DirectDraw surface—that is, **IDirect3DDevice** and **IDirectDrawSurface** were two interfaces to the same object. A given Direct3D object supported multiple 3-D device types. The **IDirect3D** interface was used to find or enumerate the device types. The **IDirect3D::EnumDevices** and **IDirect3D::FindDevice** methods identified the various device types by unique interface IDs (IIDs), which were then used to retrieve a Direct3D device interface by calling the **QueryInterface** method on a DirectDraw surface. The lifetimes of the DirectDraw surface and the Direct3D device were identical, since the same object implemented them. This architecture did not allow the programmer to change the rendering target of the Direct3D device

In DirectX 5 there are two models of Direct3D device objects. In the new model, Direct3D devices are separate objects from DirectDraw surface objects. For backward compatibility with DirectX 3 applications, the earlier model, in which Direct3D devices and DirectDraw surfaces were aggregated, is also supported. You cannot use the new DirectX 5 features with the old model. If your application calls the **QueryInterface** method on a DirectDraw surface and retrieves an **IDirect3DDevice**, it is using the old device model. You cannot call the **QueryInterface** method on a device object created in this way to retrieve an **IDirect3DDevice2** interface.

It is recommended that all applications written with the DirectX 5 SDK use the new device object model.

In DirectX 5, the new device model has the Direct3D device as a separate object from a DirectDraw surface. The **IDirect3D2** interface is used to find or enumerate the types of devices supported. However, **IDirect3D2** identifies devices by unique IDs known as class IDs in COM (CLSID). These are used to uniquely identify one of the various classes that implement a given interface. Since there are multiple Direct3D devices with different capabilities (some software based, some hardware based), but each supports the same set of interfaces, a CLSID is used to identify which type of device object we want. The CLSID obtained from **IDirect3D2::FindDevice** or **IDirect3D2::EnumDevices** is then used in a call to the **IDirect3D2::CreateDevice** method to create a device. The device objects created in this fashion support both **IDirect3DDevice** and **IDirect3DDevice2** interfaces. Unlike in DirectX 3, however, you cannot call QueryInterface on these objects to retrieve an **IDirectDrawSurface** interface. Instead, you must use the **IDirect3DDevice2::GetRenderTarget** method.

The **IDirect3DTexture** interface is not an interface to a distinct object type, but instead is another interface to a DirectDrawSurface object. The same rules for reference counts and state lifetimes that apply to **IDirect3D2** interfaces to DirectDraw objects also apply to Direct3D textures.

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

Texture Maps

You can allocate texture map surface by specifying the **DDSCAPS_TEXTURE** flag in the **ddsCaps** member of the **DDSURFACEDESC** structure passed to the **IDirectDraw2::CreateSurface** method.

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

Z-Buffers

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

An application can clear z-buffers by using the **IDirectDrawSurface3::Blt** method. The DDBLT_DEPTHFILL flag indicates that the blit clears z-buffers. If this flag is specified, the **DDBLTFX** structure passed to the **IDirectDrawSurface3::Blt** method should have its **dwFillDepth** member set to the required z-depth. If the DirectDraw device driver for a 3-D-accelerated display card is designed to provide support for z-buffer clearing in hardware, it should export the DDCAPS_BLTDEPTHFILL flag and should handle DDBLT_DEPTHFILL blits. The destination surface of a depth-fill blit must be a z-buffer.

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

RGBZ Support

DirectDraw supports the RGBZ pixel format. In RGBZ, bits that do not store colors store depth information; instead of being stored in a separate z-buffer, depth information is stored with each pixel.

The RGBZ format is particularly useful for applications that rely on emulating 3-D capabilities in software. Complicated 3-D scenes typically use many small triangles. When z-information is kept in a separate buffer, applications must access random memory locations repeatedly for each line of a triangle; once to check the z-buffer, again to write the new color value (if necessary), and again for the same area (in the common case of overlapping triangles). Applications using RGBZ pixels can perform one memory access for an entire line in a triangle, and there is no additional overhead if the hardware can drop the z-information automatically.

DirectDraw supports copying to an RGBZ surface and clearing it, but it does not contain any methods that directly exploit the depth information.

DirectDraw can copy from a source surface to a destination surface only when the surfaces have exactly the same pixel format. DirectDraw does not emulate copying from an RGBZ surface to an RGB surface.

Some hardware can copy from RGBZ surfaces to RGB surfaces. A useful feature in such copy operations is the ability to drop the z-information automatically. Drivers use the **dwSVBCaps2** member of the **DDCAPS** structure to specify that they can perform RGBZ to RGB conversions. (The **dwSVBCaps** member specifies other capabilities of system- to video-memory blits.)

Most implementations of RGBZ pixel formats support dropping the z-information on blits from system- to video memory, not on blits from video-to-video memory. Blits from system- to video-memory can often be performed asynchronously, leaving more processor time for rendering or logic. In addition, access to z-buffers is read-write intensive and therefore usually occurs in system memory. Applications that sometimes run out of space in video memory (for example, applications that use many textures) will find system- to video-memory blits useful.

Applications can use the **dwFillPixel** member of the **DDBLTFX** structure to apply color fills to RGBZ surfaces. (You cannot fill only the color or only the z-portions of an RGBZ surface—you must set both. The **dwFillPixel** member does this for you.)

Execute Buffers

In the past, all programming with Direct3D Immediate Mode was done using execute buffers. Now that the DrawPrimitive methods have been introduced, however, most new Immediate-Mode programs will not use execute buffers or the **IDirect3DExecuteBuffer** interface. For more information about the DrawPrimitive methods, see The DrawPrimitive Methods.

Execute buffers are similar to the display lists you may be familiar with if you have experience with OpenGL programming. 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. Each execute buffer is bound to a single Direct3D device.

You can create an **IDirect3DExecuteBuffer** interface 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-buffers 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.

Using Execute Buffers

As was pointed out earlier, there are two ways to use Immediate Mode: you can use the DrawPrimitive methods or you can work with execute buffers (display lists). Most developers who have never worked with Immediate Mode before will use the DrawPrimitive methods. Developers who already have an investment in code that uses execute buffers will probably continue to work with them. For more information about the DrawPrimitive methods, see **The DrawPrimitive Methods**.

Execute buffers are complex to understand and fill and are difficult to debug. On the other hand, they allow you to maximize performance. Since communicating with the driver is slow, it makes sense to perform the communication in batches—that is, by using execute buffers.

This section of the documentation describes the contents of execute buffers and how to use them.

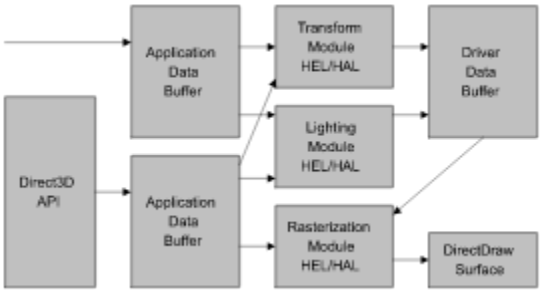
- Execute-Buffer Architecture
- Execute-Buffer Contents
- Creating an Execute Buffer
- Locking the Execute Buffer
- Filling the Execute Buffer
- Unlocking the Execute Buffer
- Executing the Execute Buffer

Execute-Buffer Architecture

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 visible. 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.

There are really two execute buffers: one for the application and one for the driver. The application data buffer is filled in by the application. It holds geometry (such as vertices and triangles) and state information (the transformation, lighting, and rasterization state) This information persists until the application explicitly changes it. The driver data buffer, on the other hand, holds the output of the transformation and lighting modules (that is, it holds transformed and lit geometry) and hands the data off to the rasterization module. There is only one of these "TL buffers" per driver. The following diagram shows the relationship of these data buffers:



You can disable the lighting module or both the lighting and transformation 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. For more information about vertex types, see Vertex Types.

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.

Execute-Buffer Contents

Execute buffers contain a list of vertices followed by stream of instructions about how to use those vertices. All of these are DWORD-aligned.

The following illustration shows the format of execute buffers.

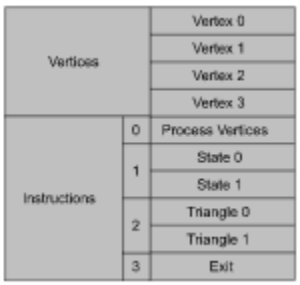
The instruction stream consists of operation codes, or *opcodes*, and the data that is operated on by those opcodes. The opcodes define how the vertex list should be lit and rendered. Direct3D opcodes are listed in the **D3DOPCODE** enumerated type. The **D3DINSTRUCTION** structure describes instructions 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.

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.

Execute Buffer Vertices

Each execute buffer contains a vertex list followed by an instruction stream. The instruction stream defines how the vertex list should be rendered; it is based on indices into the vertex list.

Although you can choose to use transformed and lit vertices (**D3DTLVERTEX**), vertices that have only been lit (**D3DLVERTEX**), or vertices that have been neither transformed nor lit (**D3DVERTEX**), you can have only one of each type of vertex in a single Direct3DExecuteBuffer object. Some execute buffers are used only to change the state of one or more of the modules in the graphics pipeline; these execute buffers do not have vertices.



For more information about the handling of vertices in execute buffers, see Vertex Types.

Execute Buffer Instructions

The vertex data in an execute buffer is followed by an instruction stream.

Each instruction is represented by:

- An instruction header
- Opcode
- Byte size
- Number of times this opcode is to be repeated
- Byte offset to first instruction

Execute buffer instructions are commands to the driver. Each instruction is identified by an operation code (opcode). All execute data is prefixed by an instruction header. Data accompanies each iteration of each opcode.

Each opcode can have multiple arguments, including multiple triangles or multiple state changes.

There are only a few main instruction types:

- **Drawing**
- **State changes**
- **Control flow**
- **Others**

Drawing Instructions

The most important of the drawing instructions defines a triangle. In a triangle, vertices are zero-based indices into the vertex list that begins the execute buffer. For more information about triangles, see Triangles.

Other important drawing instructions include line-drawing instructions (**D3DLINE**) and line-drawing instructions (**D3DPOINT**).

State-change Instructions

The system stores the state of each of the modules in the graphics pipeline until the state is overridden by an instruction in an execute buffer.

Transformation state	World, view and projection matrices
Light state	Surface material, fog, ambient lighting
Render state	Texture, antialiasing, z-buffering, and so on

Flow-control Instructions

The flow-control instructions allow you to branch on an instruction or to jump to a new position in the execute buffer, skipping or repeating instructions as necessary. This means that you can use the flow-control instructions as a kind of programming language.

The last flow-control instruction in an execute buffer must be **D3DOP_EXIT**.

Other Instructions

Some other execute-buffer instructions do not fall neatly into the other categories. These include:

Texturing	Download a texture to the device
Matrices	Download or multiply a matrix
Span, SetState	Advanced control for primitives and rendering states.

Creating an Execute Buffer

The tricky thing about creating an execute buffer is figuring out how much memory to allocate for it. There are two basic strategies for determining the correct size:

- Add the sizes of the vertices, opcodes, and data you will be putting into the buffer.
- Allocate a buffer of an arbitrary size and fill it from both ends, putting the vertices at the beginning and the opcodes at the end. When the buffer is nearly full, execute it and allocate another.

The hardware determines the size of the execute buffer. You can retrieve this size by calling the **IDirect3DDevice2::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.

After filling in **D3DEXECUTEBUFFERDESC** structure describing your execute buffer, you can call the **IDirect3DDevice::CreateExecuteBuffer** to create it.

For an example of calculating the size of the execute buffer and creating it, see Creating the Scene, in the Direct3D Execute-Buffer Tutorial.

Locking the Execute Buffer

You must lock execute buffers before you can modify them. This action prevents the driver from modifying the buffer while you are working with it.

To lock a buffer, call the **IDirect3DExecuteBuffer::Lock** method. This method takes a single parameter; a pointer to a **D3DEXECUTEBUFFERDESC** structure which, on return, specifies the actual location of the execute buffer's memory.

When working with execute buffers you need to manage three pointers: the execute buffer's start address (retrieved by **IDirect3DExecuteBuffer::Lock**), the instruction start address, and your current position in the buffer. You will use these three pointers to compute vertex offsets, instruction offsets, and the overall size of the execute buffer. When you have finished filling the execute buffer, you will use these pointers to describe the buffer to the driver; for more information about this, see Unlocking the Execute Buffer.

Filling the Execute Buffer

When you have finished filling your execute buffer, it will contain the vertices describing your model and a series of instructions about how the vertices should be interpreted. The following sections describe filling an execute buffer:

- Vertex Types
- Triangles
- Processing Vertices
- Finishing the Instructions

You can streamline the task of filling execute buffers by taking advantages of the helper macros that ship with the samples in the DirectX SDK. The D3dmacs.h header file in the Misc directory of the samples contains many useful macros that will simplify your work.

For an example of filling an execute buffer, see Filling the Execute Buffer, in the Direct3D Execute-Buffer Tutorial.

Vertex Types

Applications that only need to use part of the graphics pipeline can use specialized vertex types into their execute buffers. The following sections discuss three different vertex types:

- Transformed and Lit Vertex
- Lit Vertex
- Vertex (Model Vertex)

You can use one of the helper macros in D3dmacs.h to help you copy data into the execute buffer. This macro is **VERTEX_DATA**.

Transformed and Lit Vertex

The **D3DTLVERTEX** structure defines a transformed and lit vertex (screen coordinates with color). You should use this vertex type when the transform and lighting is not being done through Direct3D.

This vertex type may be the easiest type to use when porting existing 3-D applications to Direct3D.

Lit Vertex

The **D3DLVERTEX** structure defines an untransformed vertex (model coordinates with color). You should use this vertex type when Direct3D will not be used to provide the lighting of these vertex.

Direct3D transforms these vertices prior to rasterization. They are useful for prelit models and scenes with static light sources.

Vertex (Model Vertex)

The **D3DVERTEX** structure defines an untransformed and unlit vertex (model coordinates). You should use this vertex type when you want Direct3D to perform the transformation and the lighting prior to rasterization. This is the vertex type used by Direct3D Retained Mode. It is the best vertex to use if you wants to maximize potential hardware acceleration.

Triangles

You use the **D3DOP_TRIANGLE** opcode to insert a triangle into an execute buffer. In a triangle, vertices are zero-based indices into the vertex list that begins the execute buffer. Triangles are described by the **D3DTRIANGLE** structure.

Triangles are the only geometry type that can be processed by the rasterization module. 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 zero at the front of the viewing frustum to one 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 back facing 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.

You should be sure that your triangle data is aligned on QWORD (8-byte) boundaries. The **OP_NOP** helper macro in D3dmacs.h can help you with this alignment task. Note that if you use this macro, you must always bracket it with opening and closing braces.

Processing Vertices

After filling in the vertices in your execute buffer, you typically use the **D3DOP_PROCESSVERTICES** opcode to set the lighting and transformations for the vertices.

The **D3DPROCESSVERTICES** structure describes how the vertices should be processed. The **dwFlags** member of this structure specifies the type of vertex you are using in your execute buffer. If you are using **D3DTLVERTEX** vertices, you should specify **D3DPROCESSVERTICES_COPY** for **dwFlags**. For **D3DLVERTEX**, specify **D3DPROCESSVERTICES_TRANSFORM**. For **D3DVERTEX**, specify **D3DPROCESSVERTICES_TRANSFORMLIGHT**.

Finishing the Instructions

The last opcode in your list of instructions should be **D3DOP_EXIT**. This opcode simply signals that the system can stop processing the data.

Unlocking the Execute Buffer

When you have finished filling the execute buffer, you must unlock it. This alerts the driver that it can work with the buffer. You can unlock the buffer by calling the **IDirect3DExecuteBuffer::Unlock** method.

When the execute buffer has been unlocked, call the **IDirect3DExecuteBuffer::SetExecuteData** method to give the driver some important details about the buffer. This method takes a pointer to a **D3DEXECUTEDATA** structure. Among the information you will provide in this structure are the offsets of the vertices and instructions, which you will have been tracking ever since locking the buffer, as described in Locking the Execute Buffer.

Executing the Execute Buffer

Executing an execute buffer is a simple matter of calling the **IDirect3DDevice::Execute** method with pointers to the execute buffer and to the viewport describing the rendering target. You should always check the return value from this method to verify that it was successful.

The *dwFlags* parameter of **IDirect3DDevice::Execute** specifies whether the vertices you supply should be clipped. You should specify D3DEXECUTE_UNCLIPPED if you are using **D3DTLVERTEX** vertices and D3DEXECUTE_CLIPPED otherwise.

When you have executed an execute buffer, you can delete it. This is done simply by calling the **IDirect3DExecuteBuffer::Release** method. You could also use the **RELEASE** macro in D3dmacs.h, if you prefer.

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 SDK sample.

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 3-D 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:

- PC Hardware Accelerators
- Databases and Culling
- Batching Primitives
- Lighting Tips
- Texture Size
- Software versus Hardware
- Triangle Flags
- Clip Tests on Execution
- General Performance Tips

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 Textures
- Copy Texture-blending Mode
- Ramp Performance Tips
- Z-Buffer Performance

PC Hardware Accelerators

You will be disappointed by the performance of most of the current 3-D cards. It is unlikely that they will supply a significantly greater polygon throughput rate than unaccelerated hardware. This situation is changing quickly though, with a combination of market forces, design experience, and increasing hardware expertise.

You should be skeptical of published performance figures. They are typically peak rates, produced in ideal conditions for the renderer, and are not actually reproducible in a real-world application.

In the meantime, even if the performance is not what you might desire, many of the current hardware accelerators are still useful. For example, they typically fill polygons very quickly, particularly big polygons. The resolution of your image, the color depth at which you can run, and some special effects might also improve.

Databases and Culling

Building a reliable database of the objects in your world is the key to excellent performance in Direct3D—it is more important than improvements to rasterization or hardware.

You should maintain the lowest polygon count you can possibly manage. Design for a low polygon count, building low-poly models from the start, and add polygons if you feel that you can do so without sacrificing performance later in the development process. Try to keep the total number of polygons in the neighborhood of 2500. Remember, "the fastest polygons are the ones you don't draw."

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.

Lighting Tips

Since lights add a per-vertex cost to each rendered frame, you can achieve significant performance improvements by being careful about how you use them in your application. Most of the following tips derive from the maxim, "the fastest code is code that is never called."

- Use as few lights as possible. If you just need to bring up the overall level of lighting, use the ambient light instead of adding a new light. (It's much cheaper.)
- Directional lights are cheaper than point lights or spotlights. For directional lights, the direction to the light is fixed and doesn't need to be calculated on a per-vertex basis.
- Spotlights can be cheaper than point lights, because the area outside of the cone of light is calculated quickly. Whether or not they are cheaper depends on how much of your scene is lit by the spotlight.
- Use the range parameter to limit your lights to only the parts of the scene you need to illuminate. All the light types exit fairly early when they are out of range.
- Specular highlights almost double the cost of a light—use them only when you must. Use the D3DLIGHT_NO_SPECULAR flag in the **D3DLIGHT2** structure as often as reasonable. When defining materials you must set the specular power value to zero to turn off specular highlights for that material—simply setting the specular color to 0,0,0 is not enough.

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 256×256 are the fastest. If your application uses four 128×128 textures, for example, try to ensure that they use the same palette and place them all into one 256×256 texture. This technique also reduces the amount of texture swapping. Of course, you should not use 256×256 textures unless your application requires that much texturing because, as already mentioned, textures should be kept as small as possible.

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.

Copy Texture-blending Mode

Applications that use the ramp driver can sometimes improve performance by using the **D3DTBLEND_COPY** texture-blending mode from the **D3DTEXTUREBLEND** enumerated type. This mode is an optimization for software rasterization; for applications using a HAL, it is equivalent to the **D3DTBLEND_DECAL** texture-blending mode.

Copy mode is the simplest form of rasterization and hence the fastest. When copy mode rasterization is used, no lighting or shading is performed on the texture. The bytes from the texture are copied directly to the screen and mapped onto polygons using the texture coordinates in each vertex. Hence, when using copy mode, your application's textures must use the same pixel format as the primary surface. They must also use the same palette as the primary surface.

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. You can use D3dtest.exe to verify system performance in this case.

Copy mode implements only two rasterization options, z-buffering and chromakey transparency. The fastest mode is to simply map the texels to the polygons, with no transparency and no z-buffering. Enabling chromakey transparency accelerates the rasterization of invisible pixels because only the texture read is performed, but visible pixels will incur a slight performance degradation because of the chromakey test.

Enabling z-buffering incurs the largest performance degradation for 8 bit copy mode. When z-buffering is enabled, a 16 bit value has to be read and conditionally written per pixel. Even so, enabling z-buffering for copy mode can be faster than disabling it if the average overdraw goes over two and the scene is rendered in front-to-back polygon order.

If your scene has overdraw of less than 2 (which is very likely) you should not use z-buffering in copy mode. The only exception to this rule is if the scene complexity is very high. For example, if you have more than about 1500 rendered polygons in the scene, the sort overhead begins to get high. In that case, it may be worth considering a z-buffer again.

Direct3D is fastest when all it needs to draw is one long triangle instruction. Render state changes just get in the way of this; the longer the average triangle instruction, the better the triangle throughput. Therefore, peak sorting performance can be achieved when all the textures for a given scene are contained in only one texture map or texture page. Although this adds the restriction that no texture coordinate can be larger than 1.0, it has the performance benefit of completely avoiding texture state changes.

For normal simple scenes use one texture, one material, and sort the triangles. Use z-buffering only when the scene becomes complex.

Software versus Hardware

It isn't always obvious that you should use hardware over software renderers. Software renderers work with small polygons efficiently but are not so adept at working with big ones. They use the monochromatic model rather than the RGB model. Hardware renderers, on the other hand, are good at big polygons but less good at small ones. They use the RGB model instead of the monochromatic model. Even if you use hardware, the transformation and lighting are likely to be in software, so CPU speed is still critical to excellent performance.

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.

Clip Tests on Execution

Applications that use execute buffers 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 Retained Mode automatically uses these features to speed up its use of execute buffers.

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 256×256 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.

Z-Buffer Performance

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 scan line basis. If a scan line 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. If you render the bounding box of a complex object using z-visibility testing, you can easily discover whether the object is completely hidden. If it is hidden, you can avoid even starting to render the object. For example, imagine that the camera is in a room full of 3-D objects. Adjoining this room is a second room full of 3-D objects. The rooms are connected by an open door. If you render the first room and then draw the doorway to the second room using a z-test polygon, you may discover that the doorway is hidden by one of the objects in the first room and that you don't need to render anything at all in the second room.

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 **IDirectDrawSurface3::Lock** and **IDirectDrawSurface3::Unlock** methods, as well as between calls to the **IDirectDrawSurface3::GetDC** and **IDirectDrawSurface3::ReleaseDC** methods.)

General Performance Tips

You can follow a few general guidelines to increase the performance of your application.

- Only clear when you must.
- Minimize state changes.
- Use perspective correction only if you must.
- If you can use smaller textures, do so.
- Gracefully degrade special effects that require a disproportionate share of system resources.
- Constantly test your application's performance.
- Ensure that your application runs well both with hardware acceleration and software emulation.

Troubleshooting

This section lists common categories of problems that you may encounter when writing Direct3D programs, and what you should do to prevent them.

- Device Creation
- Nothing Visible
- Debugging
- Borland Floating-Point Initialization
- Miscellaneous

Device Creation

If your application fails during device creation, check for the following common errors:

- You must specify DDSCAPS_3DDEVICE when you create the DirectDraw surface.
- If you're using a palettized device, you must attach the palette.
- If you're using a z-buffer, you must attach it to the rendering target.
- Make sure you check the device capabilities, particularly the render depths.
- Check whether you are using system or video memory.
- Ensure that the registry has not been corrupted.

Nothing Visible

If your application runs but nothing is visible, check for the following common errors:

- Ensure that your triangles are not degenerate.
- Make sure that your index lists are internally consistent—that you don't have entries like 1, 2, 2 (which are silently dropped).
- Ensure that your triangles are not being culled.
- Make sure that your transformations are internally consistent.
- Check the viewport to be sure it will allow your triangles to be seen.
- Check the description of the execute buffer.

Debugging

Debugging a Direct3D application can be challenging. In addition to checking all the return values (a particularly important piece of advice in Direct3D programming, which is so dependent on very different hardware implementations), try the following techniques:

- Switch to debug DLLs.
- Force a software-only device, turning off hardware acceleration even when it is available.
- Force surfaces into system memory.
- Create an option to run in a window, so that you can use an integrated debugger.

The second and third options in the preceding list can help you avoid the Win16 lock which can otherwise cause your debugger to hang.

Also, try adding the following entries to WIN.INI:

```
[Direct3D]
debug=3
[DirectDraw]
debug=3
```

Borland Floating-Point Initialization

Compilers from the Borland company report floating-point exceptions in a manner that is incompatible with Direct3D. To solve this problem, you should include a _matherr() exception handler like the following:

```
// Borland floating point initialization
#include <math.h>
#include <float.h>

void initfp(void)
{
    // disable floating point exceptions
    _control87(MCW_EM,MCW_EM);
}

int _matherr(struct _exception  *e)
{
    e;                // dummy reference to catch the warning
    return 1;    // error has been handled
}
```

Miscellaneous

The following tips can help you uncover common miscellaneous errors:

- Check the memory type (system or video) for your textures.
- Verify that the current hardware can do texturing.
- Make sure that you can restore any lost surfaces.
- Always specify D3DLIGHTSTATE_MATERIAL, even in RGB mode, because it is always necessary in monochromatic mode.

Direct3D Execute-Buffer Tutorial

To create a Direct3D Immediate-Mode application based on execute buffers, you create DirectDraw and Direct3D objects, set render states, fill execute buffers, and execute those buffers.

This section includes a simple Immediate-Mode application that draws a single, rotating, Gouraud-shaded triangle. The triangle is drawn in a window whose size is fixed. For code clarity, we have chosen not to address a number of issues in this sample. For example, full-screen operation, resizing the window, and texture mapping are not included. Furthermore, we have not included some optimizations when their inclusion would have made the code more obscure. Code comments highlight the places in which we did not implement a common optimization.

Definitions, Prototypes, and Globals

This section contains the definitions, function prototypes, global variables, constants, and other structural underpinnings for the Imsample.c code sample.

Header and Includes

```
/*****************************************************************************
 *
 * File :       imsample.c
 *
 * Author :     Colin D. C. McCartney
 *
 * Date :       1/7/97
 *
 * Version :    V1.1
 *
 *****************************************************************************/

/*****************************************************************************
 *
 * Include files
 *
 *****************************************************************************/

#define  INITGUID
#include <windows.h>
#include <math.h>
#include <assert.h>
#include <ddraw.h>
#include <d3d.h>

#include "resource.h"
```

Constants in Imsample.c

```
// Class name for this application's window class.

#define WINDOW_CLASSNAME      "D3DSample1Class"

// Title for the application's window.

#define WINDOW_TITLE          "D3D Sample 1"

// String to be displayed when the application is paused.

#define PAUSED_STRING         "Paused"

// Half height of the view window.

#define HALF_HEIGHT           D3DVAL(0.5)

// Front and back clipping planes.

#define FRONT_CLIP            D3DVAL(1.0)
#define BACK_CLIP             D3DVAL(1000.0)

// Fixed window size.

#define WINDOW_WIDTH          320
#define WINDOW_HEIGHT         200

// Maximum length of the chosen device name and description of the
// chosen Direct3D device.

#define MAX_DEVICE_NAME       256
#define MAX_DEVICE_DESC       256

// Amount to rotate per frame.

#define M_PI                  3.14159265359
#define M_2PI                 6.28318530718
#define ROTATE_ANGLE_DELTA    (M_2PI / 300.0)

// Execute buffer contents

#define NUM_VERTICES          3
#define NUM_INSTRUCTIONS      6
#define NUM_STATES            7
#define NUM_PROCESSVERTICES   1
#define NUM_TRIANGLES         1
```

Macros in Imsample.c

```
// Extract the error code from an HRESULT

#define CODEFROMHRESULT(hRes) ((hRes) & 0x0000FFFF)

#ifdef _DEBUG
#define ASSERT(x)       assert(x)
#else
#define ASSERT(x)
#endif

// Used to keep the compiler from issuing warnings about any unused
// parameters.

#define USE_PARAM(x)    (x) = (x)
```

Global Variables

```
// Application instance handle (set in WinMain).

static HINSTANCE               hAppInstance              = NULL;

// Running in debug mode?

static BOOL                    fDebug                    = FALSE;

// Is the application active?

static BOOL                    fActive                   = TRUE;

// Has the application been suspended?

static BOOL                    fSuspended                = FALSE;

// DirectDraw interfaces

static LPDIRECTDRAW            lpdd                      = NULL;
static LPDIRECTDRAWSURFACE     lpddPrimary               = NULL;
static LPDIRECTDRAWSURFACE     lpddDevice                = NULL;
static LPDIRECTDRAWSURFACE     lpddZBuffer               = NULL;
static LPDIRECTDRAWPALETTE     lpddPalette               = NULL;

// Direct3D interfaces

static LPDIRECT3D              lpd3d                     = NULL;
static LPDIRECT3DDEVICE        lpd3dDevice               = NULL;
static LPDIRECT3DMATERIAL      lpd3dMaterial             = NULL;
static LPDIRECT3DMATERIAL      lpd3dBackgroundMaterial   = NULL;
static LPDIRECT3DVIEWPORT      lpd3dViewport             = NULL;
static LPDIRECT3DLIGHT         lpd3dLight                = NULL;
static LPDIRECT3DEXECUTEBUFFER lpd3dExecuteBuffer        = NULL;

// Direct3D handles

static D3DMATRIXHANDLE         hd3dWorldMatrix           = 0;
static D3DMATRIXHANDLE         hd3dViewMatrix            = 0;
static D3DMATRIXHANDLE         hd3dProjMatrix            = 0;
static D3DMATERIALHANDLE       hd3dSurfaceMaterial       = 0;
static D3DMATERIALHANDLE       hd3dBackgroundMaterial    = 0;

// Globals used for selecting the Direct3D device. They are
// globals because this makes it easy for the enumeration callback
// function to read and write from them.

static BOOL                    fDeviceFound              = FALSE;
static DWORD                   dwDeviceBitDepth          = 0;
static GUID                    guidDevice;
static char                    szDeviceName[MAX_DEVICE_NAME];
static char                    szDeviceDesc[MAX_DEVICE_DESC];
static D3DDEVICEDESC           d3dHWDeviceDesc;
```

```
static D3DDEVICEDESC          d3dSWDeviceDesc;

// The screen coordinates of the client area of the window. This
// rectangle defines the destination into which we blit to update
// the client area of the window with the results of the 3-D rendering.

static RECT                   rDstRect;

// This rectangle defines the portion of the rendering target surface
// into which we render. The top-left coordinates of this rectangle
// are always zero; the right and bottom coordinates give the size of
// the viewport.

static RECT                   rSrcRect;

// Angle of rotation of the world matrix.

static double                 dAngleOfRotation          = 0.0;

// Predefined transformations.

static D3DMATRIX d3dWorldMatrix =
{
    D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0)
};

static D3DMATRIX d3dViewMatrix =
{
    D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 5.0), D3DVAL( 1.0)
};

static D3DMATRIX d3dProjMatrix =
{
    D3DVAL( 2.0), D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 2.0), D3DVAL( 0.0), D3DVAL( 0.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL( 1.0), D3DVAL( 1.0),
    D3DVAL( 0.0), D3DVAL( 0.0), D3DVAL(-1.0), D3DVAL( 0.0)
};
```

Function Prototypes

```
static void           ReportError(HWND hwnd, int nMessage,
                                  HRESULT hRes);
static void           FatalError(HWND hwnd, int nMessage, HRESULT hRes);

static DWORD          BitDepthToFlags(DWORD dwBitDepth);
static DWORD          FlagsToBitDepth(DWORD dwFlags);

static void           SetPerspectiveProjection(LPD3DMATRIX lpd3dMatrix,
                                               double      dHalfHeight,
                                               double      dFrontClipping,
                                               double      dBackClipping);
static void           SetRotationAboutY(LPD3DMATRIX lpd3dMatrix,
                                        double      dAngleOfRotation);

static HRESULT        CreateDirect3D(HWND hwnd);
static HRESULT        ReleaseDirect3D(void);

static HRESULT        CreatePrimary(HWND hwnd);
static HRESULT        RestorePrimary(void);
static HRESULT        ReleasePrimary(void);

static HRESULT WINAPI EnumDeviceCallback(LPGUID       lpGUID,
                                         LPSTR        lpszDeviceDesc,
                                         LPSTR        lpszDeviceName,
                                         LPD3DDEVICEDESC lpd3dHWDeviceDesc,
                                         LPD3DDEVICEDESC lpd3dSWDeviceDesc,
                                         LPVOID       lpUserArg);
static HRESULT        ChooseDevice(void);

static HRESULT        CreateDevice(DWORD dwWidth, DWORD dwHeight);
static HRESULT        RestoreDevice(void);
static HRESULT        ReleaseDevice(void);

static LRESULT        RestoreSurfaces(void);

static HRESULT        FillExecuteBuffer(void);
static HRESULT        CreateScene(void);
static HRESULT        ReleaseScene(void);
static HRESULT        AnimateScene(void);

static HRESULT        UpdateViewport(void);

static HRESULT        RenderScene(void);
static HRESULT        DoFrame(void);
static void           PaintSuspended(HWND hwnd, HDC hdc);

static LRESULT        OnMove(HWND hwnd, int x, int y);
static LRESULT        OnSize(HWND hwnd, int w, int h);
static LRESULT        OnPaint(HWND hwnd, HDC hdc, LPPAINTSTRUCT lpps);
static LRESULT        OnIdle(HWND hwnd);

LRESULT CALLBACK      WndProc(HWND hwnd, UINT msg,
```

```
                        WPARAM wParam, LPARAM lParam);
int PASCAL          WinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpszCommandLine, int cmdShow);
```

Enumerating Direct3D Devices

The first thing a Direct3D application should do is enumerate the available Direct3D device drivers. The most important API element in this job is **IDirect3D2::EnumDevices**.

This section contains the ChooseDevice function that selects among the available Direct3D devices and the EnumDeviceCallback function that implements the selection mechanism.

- Enumeration Callback Function
- Enumeration Function

This sample application does not demonstrate the enumeration of display modes, which you will need to do if your application supports full-screen rendering modes. To enumerate the display modes, call the **IDirectDraw2::EnumDisplayModes** method.

Enumeration Callback Function

The EnumDeviceCallback function is invoked for each Direct3D device installed on the system. For each device we retrieve its identifying GUID, a name and description, a description of its hardware and software capabilities, and an unused user argument.

The EnumDeviceCallback function uses the following algorithm to choose an appropriate Direct3D device:

1. Discard any devices which don't match the current display depth.
2. Discard any devices which can't do Gouraud-shaded triangles.
3. If a hardware device is found which matches points one and two, use it. However, if we are running in debug mode we will skip hardware.
4. Otherwise favor Mono/Ramp mode software renderers over RGB ones; until MMX is widespread, Mono will be faster.

This callback function is invoked by the ChooseDevice enumeration function, which is described in Enumeration Function.

Note that the first parameter passed to this callback function, lpGUID, is NULL for the primary device. All other devices should have a non-NULL pointer. You should consider saving the actual GUID for the device you choose, not the pointer to the GUID, in case the pointer is accidentally corrupted.

```
static HRESULT WINAPI
EnumDeviceCallback(LPGUID          lpGUID,
                   LPSTR           lpszDeviceDesc,
                   LPSTR           lpszDeviceName,
                   LPD3DDEVICEDESC lpd3dHWDeviceDesc,
                   LPD3DDEVICEDESC lpd3dSWDeviceDesc,
                   LPVOID          lpUserArg)
{
    BOOL            fIsHardware;
    LPD3DDEVICEDESC lpd3dDeviceDesc;

        // Call the USE_PARAM macro on the unused parameter to
        // avoid compiler warnings.

    USE_PARAM(lpUserArg);

        // If there is no hardware support the color model is zero.

    fIsHardware     = (0 != lpd3dHWDeviceDesc->dcmColorModel);
    lpd3dDeviceDesc = (fIsHardware ? lpd3dHWDeviceDesc :
                                     lpd3dSWDeviceDesc);

        // If we are in debug mode and this is a hardware device,
        // skip it.

    if (fDebug && fIsHardware)
        return D3DENUMRET_OK;

        // Does the device render at the depth we want?

    if (0 == (lpd3dDeviceDesc->dwDeviceRenderBitDepth &
              dwDeviceBitDepth))
    {
```

```
            // If not, skip this device.

    return D3DENUMRET_OK;
}

    // The device must support Gouraud-shaded triangles.

if (D3DCOLOR_MONO == lpd3dDeviceDesc->dcmColorModel)
{
    if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
          D3DPSHADECAPS_COLORGOURAUDMONO))
    {
                    // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }
}
else
{
    if (!(lpd3dDeviceDesc->dpcTriCaps.dwShadeCaps &
          D3DPSHADECAPS_COLORGOURAUDRGB))
    {
                    // No Gouraud shading. Skip this device.

        return D3DENUMRET_OK;
    }
}

if (!fIsHardware && fDeviceFound &&
       (D3DCOLOR_RGB == lpd3dDeviceDesc->dcmColorModel))
{
            // If this is software RGB and we already have found
            // a software monochromatic renderer, we are not
            // interested. Skip this device.

    return D3DENUMRET_OK;
}

    // This is a device we are interested in. Save the details.

fDeviceFound = TRUE;
CopyMemory(&guidDevice, lpGUID, sizeof(GUID));
strcpy(szDeviceDesc, lpszDeviceDesc);
strcpy(szDeviceName, lpszDeviceName);
CopyMemory(&d3dHWDeviceDesc, lpd3dHWDeviceDesc,
           sizeof(D3DDEVICEDESC));
CopyMemory(&d3dSWDeviceDesc, lpd3dSWDeviceDesc,
           sizeof(D3DDEVICEDESC));

    // If this is a hardware device, we have found
    // what we are looking for.

if (fIsHardware)
    return D3DENUMRET_CANCEL;
```

```
        // Otherwise, keep looking.

    return D3DENUMRET_OK;
}
```

Enumeration Function

The ChooseDevice function invokes the EnumDeviceCallback function , which is described in Enumeration Callback Function.

```
static HRESULT
ChooseDevice(void)
{
    DDSURFACEDESC ddsd;
    HRESULT       hRes;

    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpddPrimary);

        // Since we are running in a window, we will not be changing the
        // screen depth; therefore, the pixel format of the rendering
        // target must match the pixel format of the current primary
        // surface. This means that we need to determine the pixel
        // format of the primary surface.

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    hRes = lpddPrimary->lpVtbl->GetSurfaceDesc(lpddPrimary, &ddsd);
    if (FAILED(hRes))
        return hRes;

    dwDeviceBitDepth =
        BitDepthToFlags(ddsd.ddpfPixelFormat.dwRGBBitCount);

        // Enumerate the devices and pick one.

    fDeviceFound = FALSE;
    hRes = lpd3d->lpVtbl->EnumDevices(lpd3d, EnumDeviceCallback,
                                      &fDeviceFound);
    if (FAILED(hRes))
        return hRes;

    if (!fDeviceFound)
    {
                // No suitable device was found. We cannot allow
                // device-creation to continue.

        return DDERR_NOTFOUND;
    }

    return DD_OK;
}
```

Creating Objects and Interfaces

This section contains functions that create the primary DirectDraw surface, a DirectDrawClipper object, a Direct3D object, and a Direct3DDevice.

- Creating the Primary Surface and Clipper Object
- Creating the Direct3D Object
- Creating the Direct3D Device

Creating the Primary Surface and Clipper Object

The CreatePrimary function creates the primary surface (representing the desktop) and creates and attaches a clipper object. If necessary, this function also creates a palette.

```
static HRESULT
CreatePrimary(HWND hwnd)
{
    HRESULT             hRes;
    DDSURFACEDESC       ddsd;
    LPDIRECTDRAWCLIPPER lpddClipper;
    HDC                 hdc;
    int                 i;
    PALETTEENTRY        peColorTable[256];

    ASSERT(NULL != hwnd);
    ASSERT(NULL != lpdd);
    ASSERT(NULL == lpddPrimary);
    ASSERT(NULL == lpddPalette);

    // Create the primary surface.

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize         = sizeof(ddsd);
    ddsd.dwFlags        = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddPrimary, NULL);
    if (FAILED(hRes))
        return hRes;

    // Create the clipper. We bind the application's window to the
    // clipper and attach it to the primary. This ensures that when we
    // blit from the rendering surface to the primary we don't write
    // outside the visible region of the window.

    hRes = DirectDrawCreateClipper(0, &lpddClipper, NULL);
    if (FAILED(hRes))
        return hRes;
    hRes = lpddClipper->lpVtbl->SetHWnd(lpddClipper, 0, hwnd);
    if (FAILED(hRes))
    {
        lpddClipper->lpVtbl->Release(lpddClipper);
        return hRes;
    }
    hRes = lpddPrimary->lpVtbl->SetClipper(lpddPrimary, lpddClipper);
    if (FAILED(hRes))
    {
        lpddClipper->lpVtbl->Release(lpddClipper);
        return hRes;
    }

    // We release the clipper interface after attaching it to the
    // surface because we don't need to use it again. The surface
    // holds a reference to the clipper when it has been attached.
    // The clipper will therefore be released automatically when the
```

```
// surface is released.

lpddClipper->lpVtbl->Release(lpddClipper);

// If the primary surface is palettized, the device will be, too.
// (The device surface must have the same pixel format as the
// current primary surface if we want to double buffer with
// DirectDraw.) Therefore, if the primary surface is palettized we
// need to create a palette and attach it to the primary surface
// (and to the device surface when we create it).

ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
hRes = lpddPrimary->lpVtbl->GetSurfaceDesc(lpddPrimary, &ddsd);
if (FAILED(hRes))
    return hRes;
if (ddsd.ddpfPixelFormat.dwFlags & DDPF_PALETTEINDEXED8)
{
// Initializing the palette correctly is essential. Since we are
// running in a window, we must not change the top ten and bottom
// ten static colors. Therefore, we copy them from the system
// palette and mark them as read only (D3DPAL_READONLY). The middle
// 236 entries are free for use by Direct3D so we mark them free
// (D3DPAL_FREE).

// NOTE: In order that the palette entries are correctly
// allocated it is essential that the free entries are
// also marked reserved to GDI (PC_RESERVED).

// NOTE: We don't need to specify the palette caps flag
// DDPCAPS_INITIALIZE. This flag is obsolete. CreatePalette
// must be given a valid palette-entry array and always
// initializes from it.

    hdc = GetDC(NULL);
    GetSystemPaletteEntries(hdc, 0, 256, peColorTable);
    ReleaseDC(NULL, hdc);

    for (i = 0; i < 10; i++)
        peColorTable[i].peFlags = D3DPAL_READONLY;
    for (i = 10; i < 246; i++)
        peColorTable[i].peFlags = D3DPAL_FREE | PC_RESERVED;
    for (i = 246; i < 256; i++)
        peColorTable[i].peFlags = D3DPAL_READONLY;
    hRes = lpdd->lpVtbl->CreatePalette(lpdd,
        DDPCAPS_8BIT, peColorTable, &lpddPalette, NULL);

    if (FAILED(hRes))
        return hRes;

    hRes = lpddPrimary->lpVtbl->SetPalette(lpddPrimary,
            lpddPalette);
        return hRes;
}
```

```
    return DD_OK;
}
```

Creating the Direct3D Object

The CreateDirect3D function creates the DirectDraw (Direct3D) driver objects and retrieves the COM interfaces for communicating with these objects. This function calls three crucial API elements: **DirectDrawCreate**, to create the DirectDraw object, **IDirectDraw2::SetCooperativeLevel**, to determine whether the application will run in full-screen or windowed mode, and **IDirectDraw::QueryInterface**, to retrieve a pointer to the Direct3D interface.

```
static HRESULT
CreateDirect3D(HWND hwnd)
{
    HRESULT hRes;

    ASSERT(NULL == lpdd);
    ASSERT(NULL == lpd3d);

    // Create the DirectDraw/3D driver object and get the DirectDraw
    // interface to that object.

    hRes = DirectDrawCreate(NULL, &lpdd, NULL);
    if (FAILED(hRes))
        return hRes;

    // Since we are running in a window, set the cooperative level to
    // normal. Also, to ensure that the palette is realized correctly,
    // we need to pass the window handle of the main window.

    hRes = lpdd->lpVtbl->SetCooperativeLevel(lpdd, hwnd, DDSCL_NORMAL);
    if (FAILED(hRes))
        return hRes;

        // Retrieve the Direct3D interface to the DirectDraw/3D driver
        // object.

    hRes = lpdd->lpVtbl->QueryInterface(lpdd, &IID_IDirect3D, &lpd3d);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Creating the Direct3D Device

The CreateDevice function creates an instance of the Direct3D device we chose earlier, using the specified width and height.

This function handles all aspects of the device creation, including choosing the surface-memory type, creating the device surface, creating the z-buffer (if necessary), and attaching the palette (if required). If you create a z-buffer, you must do so before creating an **IDirect3DDevice** interface.

```
static HRESULT
CreateDevice(DWORD dwWidth, DWORD dwHeight)
{
    LPD3DDEVICEDESC lpd3dDeviceDesc;
    DWORD           dwDeviceMemType;
    DWORD           dwZBufferMemType;
    DDSURFACEDESC   ddsd;
    HRESULT         hRes;
    DWORD           dwZBufferBitDepth;

    ASSERT(NULL != lpdd);
    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpddPrimary);
    ASSERT(NULL == lpddDevice);
    ASSERT(NULL == lpd3dDevice);

    // Determine the kind of memory (system or video) from which the
    // device surface should be allocated.

    if (0 != d3dHWDeviceDesc.dcmColorModel)
    {
        lpd3dDeviceDesc = &d3dHWDeviceDesc;

        // Device has a hardware rasterizer. Currently this means that
        // the device surface must be in video memory.

        dwDeviceMemType  = DDSCAPS_VIDEOMEMORY;
        dwZBufferMemType = DDSCAPS_VIDEOMEMORY;
    }
    else
    {
        lpd3dDeviceDesc = &d3dSWDeviceDesc;

        // Device has a software rasterizer. We will let DirectDraw
        // decide where the device surface resides unless we are
        // running in debug mode, in which case we will force it into
        // system memory. For a software rasterizer the z-buffer should
        // always go into system memory. A z-buffer in video memory will
        // seriously degrade the application's performance.

        dwDeviceMemType  = (fDebug ? DDSCAPS_SYSTEMMEMORY : 0);
        dwZBufferMemType = DDSCAPS_SYSTEMMEMORY;
    }

    // Create the device surface. The pixel format will be identical
    // to that of the primary surface, so we don't have to explicitly
```

```
// specify it. We do need to explicitly specify the size, memory
// type and capabilities  of the surface.

ZeroMemory(&ddsd, sizeof(ddsd));
ddsd.dwSize         = sizeof(ddsd);
ddsd.dwFlags        = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
ddsd.dwWidth        = dwWidth;
ddsd.dwHeight       = dwHeight;
ddsd.ddsCaps.dwCaps = DDSCAPS_3DDEVICE | DDSCAPS_OFFSCREENPLAIN |
                      dwDeviceMemType;
hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddDevice, NULL);
if (FAILED(hRes))
    return hRes;

// If we have created a palette, we have already determined that
// the primary surface (and hence the device surface) is palettized.
// Therefore, we should attach the palette to the device surface.
// (The palette is already attached to the primary surface.)

if (NULL != lpddPalette)
{
    hRes = lpddDevice->lpVtbl->SetPalette(lpddDevice, lpddPalette);
    if (FAILED(hRes))
        return hRes;
}

// We now determine whether or not we need a z-buffer and, if
// so, its bit depth.

if (0 != lpd3dDeviceDesc->dwDeviceZBufferBitDepth)
{
    // The device supports z-buffering. Determine the depth. We
    // select the lowest supported z-buffer depth to save memory.
    // (Accuracy is not too important for this sample.)

    dwZBufferBitDepth =
        FlagsToBitDepth(lpd3dDeviceDesc->dwDeviceZBufferBitDepth);

    // Create the z-buffer.

    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize            = sizeof(ddsd);
    ddsd.dwFlags           = DDSD_CAPS   |
                             DDSD_WIDTH  |
                             DDSD_HEIGHT |
                             DDSD_ZBUFFERBITDEPTH;
    ddsd.ddsCaps.dwCaps    = DDSCAPS_ZBUFFER | dwZBufferMemType;
    ddsd.dwWidth           = dwWidth;
    ddsd.dwHeight          = dwHeight;
    ddsd.dwZBufferBitDepth = dwZBufferBitDepth;
    hRes = lpdd->lpVtbl->CreateSurface(lpdd, &ddsd, &lpddZBuffer,
                                       NULL);
    if (FAILED(hRes))
        return hRes;
```

```
                // Attach it to the rendering target.

            hRes = lpddDevice->lpVtbl->AddAttachedSurface(lpddDevice,
                                                          lpddZBuffer);
            if (FAILED(hRes))
                return hRes;
        }

        // Now all the elements are in place: the device surface is in the
        // correct memory type; a z-buffer has been attached with the
        // correct depth and memory type; and a palette has been attached,
        // if necessary. Now we can query for the Direct3D device we chose
        // earlier.

        hRes = lpddDevice->lpVtbl->QueryInterface(lpddDevice,
                                        &guidDevice, &lpd3dDevice);
        if (FAILED(hRes))
            return hRes;

        return DD_OK;
    }
```

Creating the Scene

The CreateScene function creates the elements making up the 3-D scene. In this sample, the scene consists of a single light, the viewport, the background and surface materials, the three transformation matrices, and the execute buffer holding the state changes and drawing primitives.

```
static HRESULT
CreateScene(void)
{
    HRESULT              hRes;
    D3DMATERIAL          d3dMaterial;
    D3DLIGHT             d3dLight;
    DWORD                dwVertexSize;
    DWORD                dwInstructionSize;
    DWORD                dwExecuteBufferSize;
    D3DEXECUTEBUFFERDESC d3dExecuteBufferDesc;
    D3DEXECUTEDATA       d3dExecuteData;

    ASSERT(NULL != lpd3d);
    ASSERT(NULL != lpd3dDevice);
    ASSERT(NULL == lpd3dViewport);
    ASSERT(NULL == lpd3dMaterial);
    ASSERT(NULL == lpd3dBackgroundMaterial);
    ASSERT(NULL == lpd3dExecuteBuffer);
    ASSERT(NULL == lpd3dLight);
    ASSERT(0  == hd3dWorldMatrix);
    ASSERT(0  == hd3dViewMatrix);
    ASSERT(0  == hd3dProjMatrix);

    // Create the light.

    hRes = lpd3d->lpVtbl->CreateLight(lpd3d, &lpd3dLight, NULL);
    if (FAILED(hRes))
        return hRes;

    ZeroMemory(&d3dLight, sizeof(d3dLight));
    d3dLight.dwSize = sizeof(d3dLight);
    d3dLight.dltType = D3DLIGHT_POINT;
    d3dLight.dcvColor.dvR    = D3DVAL( 1.0);
    d3dLight.dcvColor.dvG    = D3DVAL( 1.0);
    d3dLight.dcvColor.dvB    = D3DVAL( 1.0);
    d3dLight.dcvColor.dvA    = D3DVAL( 1.0);
    d3dLight.dvPosition.dvX  = D3DVAL( 1.0);
    d3dLight.dvPosition.dvY  = D3DVAL(-1.0);
    d3dLight.dvPosition.dvZ  = D3DVAL(-1.0);
    d3dLight.dvAttenuation0  = D3DVAL( 1.0);
    d3dLight.dvAttenuation1  = D3DVAL( 0.1);
    d3dLight.dvAttenuation2  = D3DVAL( 0.0);
    hRes = lpd3dLight->lpVtbl->SetLight(lpd3dLight, &d3dLight);
    if (FAILED(hRes))
        return hRes;

    // Create the background material.
```

```
hRes = lpd3d->lpVtbl->CreateMaterial(lpd3d,
    &lpd3dBackgroundMaterial, NULL);
if (FAILED(hRes))
    return hRes;

ZeroMemory(&d3dMaterial, sizeof(d3dMaterial));
d3dMaterial.dwSize = sizeof(d3dMaterial);
d3dMaterial.dcvDiffuse.r  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.g  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.b  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.r  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.g  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.b  = D3DVAL(0.0);
d3dMaterial.dcvSpecular.r = D3DVAL(0.0);
d3dMaterial.dcvSpecular.g = D3DVAL(0.0);
d3dMaterial.dcvSpecular.b = D3DVAL(0.0);
d3dMaterial.dvPower       = D3DVAL(0.0);

// Since this is the background material, we don't want a ramp to be
// allocated. (We will not be smooth-shading the background.)

d3dMaterial.dwRampSize    = 1;

hRes = lpd3dBackgroundMaterial->lpVtbl->SetMaterial
    (lpd3dBackgroundMaterial, &d3dMaterial);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dBackgroundMaterial->lpVtbl->GetHandle
    (lpd3dBackgroundMaterial, lpd3dDevice, &hd3dBackgroundMaterial);
if (FAILED(hRes))
    return hRes;

// Create the viewport.
// The viewport parameters are set in the UpdateViewport function,
// which is called in response to WM_SIZE.

hRes = lpd3d->lpVtbl->CreateViewport(lpd3d, &lpd3dViewport, NULL);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dDevice->lpVtbl->AddViewport(lpd3dDevice, lpd3dViewport);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dViewport->lpVtbl->SetBackground(lpd3dViewport,
    hd3dBackgroundMaterial);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dViewport->lpVtbl->AddLight(lpd3dViewport, lpd3dLight);
if (FAILED(hRes))
    return hRes;

// Create the matrices.

hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
    &hd3dWorldMatrix);
if (FAILED(hRes))
```

```
    return hRes;
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dWorldMatrix,
    &d3dWorldMatrix);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
    &hd3dViewMatrix);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dViewMatrix,
    &d3dViewMatrix);
if (FAILED(hRes))
    return hRes;
hRes = lpd3dDevice->lpVtbl->CreateMatrix(lpd3dDevice,
    &hd3dProjMatrix);
if (FAILED(hRes))
    return hRes;
SetPerspectiveProjection(&d3dProjMatrix, HALF_HEIGHT, FRONT_CLIP,
    BACK_CLIP);
hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice, hd3dProjMatrix,
    &d3dProjMatrix);
if (FAILED(hRes))
    return hRes;

// Create the surface material.

hRes = lpd3d->lpVtbl->CreateMaterial(lpd3d, &lpd3dMaterial, NULL);
if (FAILED(hRes))
    return hRes;
ZeroMemory(&d3dMaterial, sizeof(d3dMaterial));
d3dMaterial.dwSize = sizeof(d3dMaterial);

// Base green with white specular.

d3dMaterial.dcvDiffuse.r  = D3DVAL(0.0);
d3dMaterial.dcvDiffuse.g  = D3DVAL(1.0);
d3dMaterial.dcvDiffuse.b  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.r  = D3DVAL(0.0);
d3dMaterial.dcvAmbient.g  = D3DVAL(0.4);
d3dMaterial.dcvAmbient.b  = D3DVAL(0.0);
d3dMaterial.dcvSpecular.r = D3DVAL(1.0);
d3dMaterial.dcvSpecular.g = D3DVAL(1.0);
d3dMaterial.dcvSpecular.b = D3DVAL(1.0);
d3dMaterial.dvPower       = D3DVAL(20.0);
d3dMaterial.dwRampSize    = 16;

hRes = lpd3dMaterial->lpVtbl->SetMaterial(lpd3dMaterial,
    &d3dMaterial);
if (FAILED(hRes))
    return hRes;

hRes = lpd3dMaterial->lpVtbl->GetHandle(lpd3dMaterial, lpd3dDevice,
    &hd3dSurfaceMaterial);
if (FAILED(hRes))
    return hRes;
```

```
    // Build the execute buffer.

    dwVertexSize      = (NUM_VERTICES        * sizeof(D3DVERTEX));
    dwInstructionSize = (NUM_INSTRUCTIONS    * sizeof(D3DINSTRUCTION)) +
                        (NUM_STATES          * sizeof(D3DSTATE))       +
                        (NUM_PROCESSVERTICES *
                                          sizeof(D3DPROCESSVERTICES))    +
                        (NUM_TRIANGLES      * sizeof(D3DTRIANGLE));
    dwExecuteBufferSize = dwVertexSize + dwInstructionSize;
    ZeroMemory(&d3dExecuteBufferDesc, sizeof(d3dExecuteBufferDesc));
    d3dExecuteBufferDesc.dwSize       = sizeof(d3dExecuteBufferDesc);
    d3dExecuteBufferDesc.dwFlags      = D3DDEB_BUFSIZE;
    d3dExecuteBufferDesc.dwBufferSize = dwExecuteBufferSize;
    hRes = lpd3dDevice->lpVtbl->CreateExecuteBuffer(lpd3dDevice,
        &d3dExecuteBufferDesc, &lpd3dExecuteBuffer, NULL);
    if (FAILED(hRes))
        return hRes;

    // Fill the execute buffer with the required vertices, state
    // instructions and drawing primitives.

    hRes = FillExecuteBuffer();
    if (FAILED(hRes))
        return hRes;

    // Set the execute data so Direct3D knows how many vertices are in
    // the buffer and where the instructions start.

    ZeroMemory(&d3dExecuteData, sizeof(d3dExecuteData));
    d3dExecuteData.dwSize = sizeof(d3dExecuteData);
    d3dExecuteData.dwVertexCount       = NUM_VERTICES;
    d3dExecuteData.dwInstructionOffset = dwVertexSize;
    d3dExecuteData.dwInstructionLength = dwInstructionSize;
    hRes = lpd3dExecuteBuffer->lpVtbl->SetExecuteData
        (lpd3dExecuteBuffer, &d3dExecuteData);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Filling the Execute Buffer

The **FillExecuteBuffer** function fills the single execute buffer used in this sample with all the vertices, transformations, light and render states, and drawing primitives necessary to draw our triangle.

The method shown here is not the most efficient way of organizing the execute buffer. For best performance you should minimize state changes. In this sample we submit the execute buffer for each frame in the animation loop and no state in the buffer is modified. The only thing we modify is the world matrix (its contents—not its handle). Therefore, it would be more efficient to extract all the static state instructions into a separate execute buffer which we would issue once only at startup and, from then on, simply execute a second execute buffer with vertices and triangles.

However, because this sample is more concerned with clarity than performance, it uses only one execute buffer_dx5_execute_buffer_glos and resubmits it in its entirety for each frame.

```
static HRESULT
FillExecuteBuffer(void)
{
    HRESULT              hRes;
    D3DEXECUTEBUFFERDESC d3dExeBufDesc;
    LPD3DVERTEX          lpVertex;
    LPD3DINSTRUCTION     lpInstruction;
    LPD3DPROCESSVERTICES lpProcessVertices;
    LPD3DTRIANGLE        lpTriangle;
    LPD3DSTATE           lpState;

    ASSERT(NULL != lpd3dExecuteBuffer);
    ASSERT(0  != hd3dSurfaceMaterial);
    ASSERT(0  != hd3dWorldMatrix);
    ASSERT(0  != hd3dViewMatrix);
    ASSERT(0  != hd3dProjMatrix);

    // Lock the execute buffer.

    ZeroMemory(&d3dExeBufDesc, sizeof(d3dExeBufDesc));
    d3dExeBufDesc.dwSize = sizeof(d3dExeBufDesc);
    hRes = lpd3dExecuteBuffer->lpVtbl->Lock(lpd3dExecuteBuffer,
        &d3dExeBufDesc);
    if (FAILED(hRes))
        return hRes;

    // For purposes of illustration, we fill the execute buffer by
    // casting a pointer to the execute buffer to the appropriate data
    // structures.

    lpVertex = (LPD3DVERTEX)d3dExeBufDesc.lpData;

    // First vertex.

    lpVertex->dvX  = D3DVAL( 0.0); // Position in model coordinates
    lpVertex->dvY  = D3DVAL( 1.0);
    lpVertex->dvZ  = D3DVAL( 0.0);
    lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
    lpVertex->dvNY = D3DVAL( 0.0);
    lpVertex->dvNZ = D3DVAL(-1.0);
```

```
lpVertex->dvTU = D3DVAL( 0.0); // Texture coordinates (not used)
lpVertex->dvTV = D3DVAL( 1.0);
lpVertex++;

// Second vertex.

lpVertex->dvX  = D3DVAL( 1.0); // Position in model coordinates
lpVertex->dvY  = D3DVAL(-1.0);
lpVertex->dvZ  = D3DVAL( 0.0);
lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
lpVertex->dvNY = D3DVAL( 0.0);
lpVertex->dvNZ = D3DVAL(-1.0);
lpVertex->dvTU = D3DVAL( 1.0); // Texture coordinates (not used)
lpVertex->dvTV = D3DVAL( 1.0);
lpVertex++;

// Third vertex.

lpVertex->dvX  = D3DVAL(-1.0); // Position in model coordinates
lpVertex->dvY  = D3DVAL(-1.0);
lpVertex->dvZ  = D3DVAL( 0.0);
lpVertex->dvNX = D3DVAL( 0.0); // Normalized illumination normal
lpVertex->dvNY = D3DVAL( 0.0);
lpVertex->dvNZ = D3DVAL(-1.0);
lpVertex->dvTU = D3DVAL( 1.0); // Texture coordinates (not used)
lpVertex->dvTV = D3DVAL( 0.0);
lpVertex++;

// Transform state - world, view and projection.

lpInstruction = (LPD3DINSTRUCTION)lpVertex;
lpInstruction->bOpcode = D3DOP_STATETRANSFORM;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 3U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_WORLD;
lpState->dwArg[0] = hd3dWorldMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_VIEW;
lpState->dwArg[0] = hd3dViewMatrix;
lpState++;
lpState->dtstTransformStateType = D3DTRANSFORMSTATE_PROJECTION;
lpState->dwArg[0] = hd3dProjMatrix;
lpState++;

// Lighting state.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_STATELIGHT;
lpInstruction->bSize   = sizeof(D3DSTATE);
lpInstruction->wCount  = 2U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->dlstLightStateType = D3DLIGHTSTATE_MATERIAL;
```

```
lpState->dwArg[0] = hd3dSurfaceMaterial;
lpState++;
lpState->dlstLightStateType = D3DLIGHTSTATE_AMBIENT;
lpState->dwArg[0] = RGBA_MAKE(128, 128, 128, 128);
lpState++;

// Render state.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_STATERENDER;
lpInstruction->bSize = sizeof(D3DSTATE);
lpInstruction->wCount = 3U;
lpInstruction++;
lpState = (LPD3DSTATE)lpInstruction;
lpState->drstRenderStateType = D3DRENDERSTATE_FILLMODE;
lpState->dwArg[0] = D3DFILL_SOLID;
lpState++;
lpState->drstRenderStateType = D3DRENDERSTATE_SHADEMODE;
lpState->dwArg[0] = D3DSHADE_GOURAUD;
lpState++;
lpState->drstRenderStateType = D3DRENDERSTATE_DITHERENABLE;
lpState->dwArg[0] = TRUE;
lpState++;

// The D3DOP_PROCESSVERTICES instruction tells the driver what to
// do with the vertices in the buffer. In this sample we want
// Direct3D to perform the entire pipeline on our behalf, so
// the instruction is D3DPROCESSVERTICES_TRANSFORMLIGHT.

lpInstruction = (LPD3DINSTRUCTION)lpState;
lpInstruction->bOpcode = D3DOP_PROCESSVERTICES;
lpInstruction->bSize   = sizeof(D3DPROCESSVERTICES);
lpInstruction->wCount  = 1U;
lpInstruction++;
lpProcessVertices = (LPD3DPROCESSVERTICES)lpInstruction;
lpProcessVertices->dwFlags    = D3DPROCESSVERTICES_TRANSFORMLIGHT;
lpProcessVertices->wStart     = 0U;           // First source vertex
lpProcessVertices->wDest      = 0U;
lpProcessVertices->dwCount    = NUM_VERTICES; // Number of vertices
lpProcessVertices->dwReserved = 0;
lpProcessVertices++;

// Draw the triangle.

lpInstruction = (LPD3DINSTRUCTION)lpProcessVertices;
lpInstruction->bOpcode = D3DOP_TRIANGLE;
lpInstruction->bSize   = sizeof(D3DTRIANGLE);
lpInstruction->wCount  = 1U;
lpInstruction++;
lpTriangle = (LPD3DTRIANGLE)lpInstruction;
lpTriangle->wV1    = 0U;
lpTriangle->wV2    = 1U;
lpTriangle->wV3    = 2U;
lpTriangle->wFlags = D3DTRIFLAG_EDGEENABLETRIANGLE;
lpTriangle++;
```

```
    // Stop execution of the buffer.

    lpInstruction = (LPD3DINSTRUCTION)lpTriangle;
    lpInstruction->bOpcode = D3DOP_EXIT;
    lpInstruction->bSize   = 0;
    lpInstruction->wCount  = 0U;

    // Unlock the execute buffer.

    lpd3dExecuteBuffer->lpVtbl->Unlock(lpd3dExecuteBuffer);

    return DD_OK;
}
```

Animating the Scene

The animation in this sample is simply a rotation about the y-axis. All we need to do is build a rotation matrix and set the world matrix to that new rotation matrix.

We don't need to modify the execute buffer in any way to perform this rotation. We simply set the matrix and resubmit the execute buffer.

```
static HRESULT
AnimateScene(void)
{
    HRESULT hRes;

    ASSERT(NULL != lpd3dDevice);
    ASSERT(0  != hd3dWorldMatrix);

    // We rotate the triangle by setting the world transform to a
    // rotation matrix.

    SetRotationAboutY(&d3dWorldMatrix, dAngleOfRotation);
    dAngleOfRotation += ROTATE_ANGLE_DELTA;
    hRes = lpd3dDevice->lpVtbl->SetMatrix(lpd3dDevice,
        hd3dWorldMatrix, &d3dWorldMatrix);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Rendering

This section contains functions that render the entire scene and render a single frame.

Rendering the Scene

The RenderScene function renders the 3-D scene, just as you might suspect. The fundamental task performed by this function is submitting the single execute buffer used by this sample. However, the function also clears the back and z-buffers and demarcates the start and end of the scene (which in this case is a single execute).

When you clear the back and z-buffers, it's safe to specify the z-buffer clear flag even if we don't have an attached z-buffer. Direct3D will simply discard the flag if no z-buffer is being used.

For maximum efficiency we only want to clear those regions of the device surface and z-buffer which we actually rendered to in the last frame. This is the purpose of the array of rectangles and count passed to this function. It is possible to query Direct3D for the regions of the device surface that were rendered to by that execute. The application can then accumulate those rectangles and clear only those regions. However this is a very simple sample and so, for simplicity, we will just clear the entire device surface and z-buffer. You should probably implement a more efficient clearing mechanism in your application.

The RenderScene function must be called once and once only for every frame of animation. If you have multiple execute buffers comprising a single frame you must have one call to the **IDirect3DDevice2::BeginScene** method before submitting those execute buffers. If you have more than one device being rendered in a single frame, (for example, a rear-view mirror in a racing game), call the **IDirect3DDevice::BeginScene** and **IDirect3DDevice2::EndScene** methods once for each device.

When the RenderScene function returns DD_OK, the scene will have been rendered and the device surface will hold the contents of the rendering.

```
static HRESULT
RenderScene(void)
{
    HRESULT hRes;
    D3DRECT d3dRect;

    ASSERT(NULL != lpd3dViewport);
    ASSERT(NULL != lpd3dDevice);
    ASSERT(NULL != lpd3dExecuteBuffer);

    // Clear both back and z-buffer.

    d3dRect.lX1 = rSrcRect.left;
    d3dRect.lX2 = rSrcRect.right;
    d3dRect.lY1 = rSrcRect.top;
    d3dRect.lY2 = rSrcRect.bottom;
    hRes = lpd3dViewport->lpVtbl->Clear(lpd3dViewport, 1, &d3dRect,
        D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER);
    if (FAILED(hRes))
        return hRes;

    // Start the scene.

    hRes = lpd3dDevice->lpVtbl->BeginScene(lpd3dDevice);
    if (FAILED(hRes))
        return hRes;

    // Submit the execute buffer.
```

```
    // We want Direct3D to clip the data on our behalf so we specify
    // D3DEXECUTE_CLIPPED.

    hRes = lpd3dDevice->lpVtbl->Execute(lpd3dDevice, lpd3dExecuteBuffer,
        lpd3dViewport, D3DEXECUTE_CLIPPED);
    if (FAILED(hRes))
    {
        lpd3dDevice->lpVtbl->EndScene(lpd3dDevice);
        return hRes;
    }

        // End the scene.

    hRes = lpd3dDevice->lpVtbl->EndScene(lpd3dDevice);
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Rendering a Single Frame

The DoFrame function renders and shows a single frame. This involves rendering the scene and blitting the result to the client area of the application window on the primary surface.

This function handles lost surfaces by attempting to restore the application's surfaces and then retrying the rendering. It is called by the OnMove function (discussed in Redrawing on Window Movement), the OnSize function (discussed in Redrawing on Window Resizing), and the OnPaint function (discussed in Repainting the Client Area).

```
static HRESULT
DoFrame(void)
{
    HRESULT hRes;

    // We keeping trying until we succeed or we fail for a reason
    // other than DDERR_SURFACELOST.

    while (TRUE)
    {
        hRes = RenderScene();
        if (SUCCEEDED(hRes))
        {
            hRes = lpddPrimary->lpVtbl->Blt(lpddPrimary, &rDstRect,
                lpddDevice, &rSrcRect, DDBLT_WAIT, NULL);
            if (SUCCEEDED(hRes))  // If it worked.
                return hRes;
        }
        while (DDERR_SURFACELOST == hRes)  // Restore lost surfaces
            hRes = RestoreSurfaces();
        if (FAILED(hRes))  // handle other failure cases
            return hRes;
    }
}
```

Working with Matrices

This section contains two functions that work with matrices: the SetPerspectiveProjection function, which sets a given matrix to the appropriate values for the front and back clipping plane, and the SetRotationAboutY function, which sets a matrix to a rotation about the y-axis.

- Setting the Perspective Transformation
- Setting a Rotation Transformation

Setting the Perspective Transformation

The SetPerspectiveProjection function sets the given matrix to a perspective transform for the given half-height and front- and back-clipping planes. This function is called as part of the CreateScene function, documented in Creating the Scene.

```
static void
SetPerspectiveProjection(LPD3DMATRIX lpd3dMatrix,
                         double      dHalfHeight,
                         double      dFrontClipping,
                         double      dBackClipping)
{
    double dTmp1;
    double dTmp2;

    ASSERT(NULL != lpd3dMatrix);

    dTmp1 = dHalfHeight / dFrontClipping;
    dTmp2 = dBackClipping / (dBackClipping - dFrontClipping);

    lpd3dMatrix->_11 =  D3DVAL(2.0);
    lpd3dMatrix->_12 =  D3DVAL(0.0);
    lpd3dMatrix->_13 =  D3DVAL(0.0);
    lpd3dMatrix->_14 =  D3DVAL(0.0);
    lpd3dMatrix->_21 =  D3DVAL(0.0);
    lpd3dMatrix->_22 =  D3DVAL(2.0);
    lpd3dMatrix->_23 =  D3DVAL(0.0);
    lpd3dMatrix->_24 =  D3DVAL(0.0);
    lpd3dMatrix->_31 =  D3DVAL(0.0);
    lpd3dMatrix->_32 =  D3DVAL(0.0);
    lpd3dMatrix->_33 =  D3DVAL(dTmp1 * dTmp2);
    lpd3dMatrix->_34 =  D3DVAL(dTmp1);
    lpd3dMatrix->_41 =  D3DVAL(0.0);
    lpd3dMatrix->_42 =  D3DVAL(0.0);
    lpd3dMatrix->_43 =  D3DVAL(-dHalfHeight * dTmp2);
    lpd3dMatrix->_44 =  D3DVAL(0.0);
}
```

Setting a Rotation Transformation

The SetRotationAboutY function sets the given matrix to a rotation about the y-axis, using the specified number of radians. This function is called as part of the AnimateScene function, documented in Animating the Scene.

```
static void
SetRotationAboutY(LPD3DMATRIX lpd3dMatrix, double dAngleOfRotation)
{
    D3DVALUE dvCos;
    D3DVALUE dvSin;

    ASSERT(NULL != lpd3dMatrix);

    dvCos = D3DVAL(cos(dAngleOfRotation));
    dvSin = D3DVAL(sin(dAngleOfRotation));

    lpd3dMatrix->_11 =  dvCos;
    lpd3dMatrix->_12 =  D3DVAL(0.0);
    lpd3dMatrix->_13 = -dvSin;
    lpd3dMatrix->_14 =  D3DVAL(0.0);
    lpd3dMatrix->_21 =  D3DVAL(0.0);
    lpd3dMatrix->_22 =  D3DVAL(1.0);
    lpd3dMatrix->_23 =  D3DVAL(0.0);
    lpd3dMatrix->_24 =  D3DVAL(0.0);
    lpd3dMatrix->_31 =  dvSin;
    lpd3dMatrix->_32 =  D3DVAL(0.0);
    lpd3dMatrix->_33 =  dvCos;
    lpd3dMatrix->_34 =  D3DVAL(0.0);
    lpd3dMatrix->_41 =  D3DVAL(0.0);
    lpd3dMatrix->_42 =  D3DVAL(0.0);
    lpd3dMatrix->_43 =  D3DVAL(0.0);
    lpd3dMatrix->_44 =  D3DVAL(1.0);
}
```

Restoring and Redrawing

This section contains functions that restore objects and surfaces that may have been lost while the application is running.

- Restoring the Direct3D Device
- Restoring the Primary Surface
- Restoring All Surfaces
- Redrawing on Window Movement
- Redrawing on Window Resizing
- Repainting the Client Area
- Updating the Viewport

Restoring the Direct3D Device

The RestoreDevice function restores lost video memory for the device surface and z-buffer.

```
static HRESULT
RestoreDevice(void)
{
    HRESULT hRes;

    if (NULL != lpddZBuffer)
    {
        hRes = lpddZBuffer->lpVtbl->Restore(lpddZBuffer);
        if (FAILED(hRes))
            return hRes;
    }

    if (NULL != lpddDevice)
    {
        hRes = lpddDevice->lpVtbl->Restore(lpddDevice);
        if (FAILED(hRes))
            return hRes;
    }

    return DD_OK;
}
```

Restoring the Primary Surface

The RestorePrimary function attempts to restore the video memory allocated for the primary surface. This function will be invoked by a DirectX function returning DDERR_SURFACELOST due to a mode switch or full-screen DOS box invalidating video memory.

```
static HRESULT
RestorePrimary(void)
{
    ASSERT(NULL != lpddPrimary);

    return lpddPrimary->lpVtbl->Restore(lpddPrimary);
}
```

Restoring All Surfaces

The RestoreSurfaces function attempts to restore all the surfaces used by the application.

```
static LRESULT
RestoreSurfaces(void)
{
    HRESULT hRes;

    hRes = RestorePrimary();
    if (FAILED(hRes))
        return hRes;

    hRes = RestoreDevice();
    if (FAILED(hRes))
        return hRes;

    return DD_OK;
}
```

Redrawing on Window Movement

```
static LRESULT
OnMove(HWND hwnd, int x, int y)
{
    int     xDelta;
    int     yDelta;
    HRESULT hRes;

    // No action if the device has not yet been created or if we are
    // suspended.

    if ((NULL != lpd3dDevice) && !fSuspended)
    {
        // Update the destination rectangle for the new client position.

        xDelta = x - rDstRect.left;
        yDelta = y - rDstRect.top;

        rDstRect.left   += xDelta;
        rDstRect.top    += yDelta;
        rDstRect.right  += xDelta;
        rDstRect.bottom += yDelta;

        // Repaint the client area.

        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Redrawing on Window Resizing

```
static LRESULT
OnSize(HWND hwnd, int w, int h)
{
    HRESULT       hRes;
    DDSURFACEDESC ddsd;

        // Nothing to do if we are suspended.

    if (!fSuspended)
    {
        // Update the source and destination rectangles (used by the
        // blit that shows the rendering in the client area).

        rDstRect.right  = rDstRect.left + w;
        rDstRect.bottom = rDstRect.top  + h;
        rSrcRect.right  = w;
        rSrcRect.bottom = h;

        if (NULL != lpd3dDevice)
        {
            // Although we already have a device, we need to be sure it
            // is big enough for the new window client size.

            // Because the window in this sample has a fixed size, it
            // should never be necessary to handle this case. This code
            // will be useful when we make the application resizable.

            ZeroMemory(&ddsd, sizeof(ddsd));
            ddsd.dwSize = sizeof(ddsd);
            hRes = lpddDevice->lpVtbl->GetSurfaceDesc(lpddDevice,
                &ddsd);
            if (FAILED(hRes))
            {
                FatalError(hwnd, IDS_ERRMSG_DEVICESIZE, hRes);
                return 0L;
            }

            if ((w > (int)ddsd.dwWidth) || (h > (int)ddsd.dwHeight))
            {
                // The device is too small. We need to shut it down
                // and rebuild it.

                // Execute buffers are bound to devices, so when
                // we release the device we must release the execute
                // buffer.

                ReleaseScene();
                ReleaseDevice();
            }
        }

        if (NULL == lpd3dDevice)
```

```
        {
            // No Direct3D device yet. This is either because this is
            // the first time through the loop or because we discarded
            // the existing device because it was not big enough for the
            // new window client size.

            hRes = CreateDevice((DWORD)w, (DWORD)h);
            if (FAILED(hRes))
            {
                FatalError(hwnd, IDS_ERRMSG_CREATEDEVICE, hRes);
                return 0L;
            }
            hRes = CreateScene();
            if (FAILED(hRes))
            {
                FatalError(hwnd, IDS_ERRMSG_BUILDSCENE, hRes);
                return 0L;
            }
        }

        hRes = UpdateViewport();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_UPDATEVIEWPORT, hRes);
            return 0L;
        }

        // Render at the new size and show the results in the window's
        // client area.

        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Repainting the Client Area

The OnPaint function repaints the client area, when required. Notice that it calls the DoFrame function to do much of the work, even though DoFrame re-renders the scene as well as blitting the result to the primary surface. Although the re-rendering is not necessary, for this simple sample this inefficiency does not matter. In your application, you should re-render only when the scene changes.

For more information about the DoFrame function, see Rendering a Single Frame.

```
static LRESULT
OnPaint(HWND hwnd, HDC hdc, LPPAINTSTRUCT lpps)
{
    HRESULT hRes;

    USE_PARAM(lpps);

    if (fActive && !fSuspended && (NULL != lpd3dDevice))
    {
        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }
    else
    {
        // Show the suspended image if we are not active or suspended or
        // if we have not yet created the device.

        PaintSuspended(hwnd, hdc);
    }

    return 0L;
}
```

Updating the Viewport

The UpdateViewport function updates the viewport in response to a change in window size. This ensures that we render at a resolution that matches the client area of the target window.

```
static HRESULT
UpdateViewport(void)
{
    D3DVIEWPORT d3dViewport;

    ASSERT(NULL != lpd3dViewport);

    ZeroMemory(&d3dViewport, sizeof(d3dViewport));
    d3dViewport.dwSize   = sizeof(d3dViewport);
    d3dViewport.dwX      = 0;
    d3dViewport.dwY      = 0;
    d3dViewport.dwWidth  = (DWORD)rSrcRect.right;
    d3dViewport.dwHeight = (DWORD)rSrcRect.bottom;
    d3dViewport.dvScaleX = D3DVAL((float)d3dViewport.dwWidth / 2.0);
    d3dViewport.dvScaleY = D3DVAL((float)d3dViewport.dwHeight / 2.0);
    d3dViewport.dvMaxX   = D3DVAL(1.0);
    d3dViewport.dvMaxY   = D3DVAL(1.0);
    return lpd3dViewport->lpVtbl->SetViewport(lpd3dViewport,
        &d3dViewport);
}
```

Releasing Objects

This section contains functions that release objects when they are no longer needed.

- Releasing the Direct3D Object
- Releasing the Direct3D Device
- Releasing the Primary Surface
- Releasing the Objects in the Scene

Releasing the Direct3D Object

The ReleaseDirect3D function releases the DirectDraw (Direct3D) driver object.

```
static HRESULT
ReleaseDirect3D(void)
{
    if (NULL != lpd3d)
    {
        lpd3d->lpVtbl->Release(lpd3d);
        lpd3d = NULL;
    }
    if (NULL != lpdd)
    {
        lpdd->lpVtbl->Release(lpdd);
        lpdd = NULL;
    }

    return DD_OK;
}
```

Releasing the Direct3D Device

The ReleaseDevice function releases the Direct3D device and its associated surfaces.

```
static HRESULT
ReleaseDevice(void)
{
    if (NULL != lpd3dDevice)
    {
        lpd3dDevice->lpVtbl->Release(lpd3dDevice);
        lpd3dDevice = NULL;
    }
    if (NULL != lpddZBuffer)
    {
        lpddZBuffer->lpVtbl->Release(lpddZBuffer);
        lpddZBuffer = NULL;
    }
    if (NULL != lpddDevice)
    {
        lpddDevice->lpVtbl->Release(lpddDevice);
        lpddDevice = NULL;
    }

    return DD_OK;
}
```

Releasing the Primary Surface

The ReleasePrimary function releases the primary surface and its attached clipper and palette.

```
static HRESULT
ReleasePrimary(void)
{
    if (NULL != lpddPalette)
    {
        lpddPalette->lpVtbl->Release(lpddPalette);
        lpddPalette = NULL;
    }
    if (NULL != lpddPrimary)
    {
        lpddPrimary->lpVtbl->Release(lpddPrimary);
        lpddPrimary = NULL;
    }

    return DD_OK;
}
```

Releasing the Objects in the Scene

The ReleaseScene function releases all the objects making up the 3-D scene.

```
static HRESULT
ReleaseScene(void)
{
    if (NULL != lpd3dExecuteBuffer)
    {
        lpd3dExecuteBuffer->lpVtbl->Release(lpd3dExecuteBuffer);
        lpd3dExecuteBuffer = NULL;
    }
    if (NULL != lpd3dBackgroundMaterial)
    {
        lpd3dBackgroundMaterial->
            lpVtbl->Release(lpd3dBackgroundMaterial);
        lpd3dBackgroundMaterial = NULL;
    }
    if (NULL != lpd3dMaterial)
    {
        lpd3dMaterial->lpVtbl->Release(lpd3dMaterial);
        lpd3dMaterial = NULL;
    }
    if (0 != hd3dWorldMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dWorldMatrix);
        hd3dWorldMatrix = 0;
    }
    if (0 != hd3dViewMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dViewMatrix);
        hd3dViewMatrix = 0;
    }
    if (0 != hd3dProjMatrix)
    {
        lpd3dDevice->lpVtbl->DeleteMatrix(lpd3dDevice, hd3dProjMatrix);
        hd3dProjMatrix = 0;
    }
    if (NULL != lpd3dLight)
    {
        lpd3dLight->lpVtbl->Release(lpd3dLight);
        lpd3dLight = NULL;
    }
    if (NULL != lpd3dViewport)
    {
        lpd3dViewport->lpVtbl->Release(lpd3dViewport);
        lpd3dViewport = NULL;
    }

    return DD_OK;
}
```

Error Checking

This section contains functions that help you check for and report errors.

- Checking for Active Status
- Reporting Standard Errors
- Reporting Fatal Errors
- Displaying a Notification String

Checking for Active Status

```
static LRESULT
OnIdle(HWND hwnd)
{
    HRESULT hRes;

    // Only animate if we are the foreground app, we aren't suspended,
    // and we have completed initialization.

    if (fActive && !fSuspended && (NULL != lpd3dDevice))
    {
        hRes = AnimateScene();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_ANIMATESCENE, hRes);
            return 0L;
        }

        hRes = DoFrame();
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_RENDERSCENE, hRes);
            return 0L;
        }
    }

    return 0L;
}
```

Reporting Standard Errors

The ReportError function displays a message box to report an error.

```
static void
ReportError(HWND hwnd, int nMessage, HRESULT hRes)
{
    HDC  hdc;
    char szBuffer[256];
    char szMessage[128];
    char szError[128];
    int  nStrID;

    // Turn the animation loop off.

    fSuspended = TRUE;

    // Get the high level error message.

    LoadString(hAppInstance, nMessage, szMessage, sizeof(szMessage));

    // We issue sensible error messages for common run time errors. For
    // errors which are internal or coding errors we simply issue an
    // error number (they should never occur).

    switch (hRes)
    {
        case DDERR_EXCEPTION:        nStrID = IDS_ERR_EXCEPTION;
break;
        case DDERR_GENERIC:          nStrID = IDS_ERR_GENERIC;
break;
        case DDERR_OUTOFMEMORY:      nStrID = IDS_ERR_OUTOFMEMORY;
break;
        case DDERR_OUTOFVIDEOMEMORY: nStrID = IDS_ERR_OUTOFVIDEOMEMORY;
break;
        case DDERR_SURFACEBUSY:      nStrID = IDS_ERR_SURFACEBUSY;
break;
        case DDERR_SURFACELOST:      nStrID = IDS_ERR_SURFACELOST;
break;
        case DDERR_WRONGMODE:        nStrID = IDS_ERR_WRONGMODE;
break;
        default:                     nStrID = IDS_ERR_INTERNALERROR;
break;
    }
    LoadString(hAppInstance, nStrID, szError, sizeof(szError));

    // Show the "paused" display.

    hdc = GetDC(hwnd);
    PaintSuspended(hwnd, hdc);
    ReleaseDC(hwnd, hdc);

    // Convert the error code into a string.

    wsprintf(szBuffer, "%s\n%s (Error #%d)", szMessage, szError,
```

```
        CODEFROMHRESULT(hRes));
    MessageBox(hwnd, szBuffer, WINDOW_TITLE, MB_OK | MB_APPLMODAL);
    fSuspended = FALSE;
}
```

Reporting Fatal Errors

The FatalError function displays a message box to report an error message and then destroys the window. The function does not perform any clean-up; this is done when the application receives the WM_DESTROY message sent by the **DestroyWindow** function.

```
static void
FatalError(HWND hwnd, int nMessage, HRESULT hRes)
{
    ReportError(hwnd, nMessage, hRes);
    fSuspended = TRUE;

    DestroyWindow(hwnd);
}
```

Displaying a Notification String

The PaintSuspended function draws a notification string in the client area whenever the application is suspended—for example, when it is in the background or is handling an error.

```
static void
PaintSuspended(HWND hwnd, HDC hdc)
{
    HPEN     hOldPen;
    HBRUSH   hOldBrush;
    COLORREF crOldTextColor;
    int      oldMode;
    int      x;
    int      y;
    SIZE     size;
    RECT     rect;
    int      nStrLen;

    // Black background.

    hOldPen   = SelectObject(hdc, GetStockObject(NULL_PEN));
    hOldBrush = SelectObject(hdc, GetStockObject(BLACK_BRUSH));

    // White text.

    oldMode = SetBkMode(hdc, TRANSPARENT);
    crOldTextColor = SetTextColor(hdc, RGB(255, 255, 255));

    GetClientRect(hwnd, &rect);

    // Clear the client area.

    Rectangle(hdc, rect.left, rect.top, rect.right + 1, rect.bottom + 1);

    // Draw the string centered in the client area.

    nStrLen = strlen(PAUSED_STRING);
    GetTextExtentPoint32(hdc, PAUSED_STRING, nStrLen, &size);
    x = (rect.right  - size.cx) / 2;
    y = (rect.bottom - size.cy) / 2;
    TextOut(hdc, x, y, PAUSED_STRING, nStrLen);

    SetTextColor(hdc, crOldTextColor);
    SetBkMode(hdc, oldMode);

    SelectObject(hdc, hOldBrush);
    SelectObject(hdc, hOldPen);
}
```

Converting Bit Depths

This section contains functions that convert bit depths into flags and vice versa.

Converting a Bit Depth into a Flag

The BitDepthToFlags function is used by the ChooseDevice enumeration function to convert a bit depth into the appropriate DirectDraw bit depth flag. For more information, see Enumeration Function

```
static DWORD
BitDepthToFlags(DWORD dwBitDepth)
{
    switch (dwBitDepth)
    {
        case  1: return DDBD_1;
        case  2: return DDBD_2;
        case  4: return DDBD_4;
        case  8: return DDBD_8;
        case 16: return DDBD_16;
        case 24: return DDBD_24;
        case 32: return DDBD_32;
        default: return 0;
    }
}
```

Converting a Flag into a Bit Depth

The FlagsToBitDepth function is used by the CreateDevice function to convert bit-depth flags to an actual bit count. It selects the smallest bit count in the mask if more than one flag is present. For more information, see Creating the Direct3D Device.

```
static DWORD
FlagsToBitDepth(DWORD dwFlags)
{
    if (dwFlags & DDBD_1)
        return 1;
    else if (dwFlags & DDBD_2)
        return 2;
    else if (dwFlags & DDBD_4)
        return 4;
    else if (dwFlags & DDBD_8)
        return 8;
    else if (dwFlags & DDBD_16)
        return 16;
    else if (dwFlags & DDBD_24)
        return 24;
    else if (dwFlags & DDBD_32)
        return 32;
    else
        return 0;
}
```

Main Window Procedure

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC         hdc;
    PAINTSTRUCT ps;
    LRESULT     lResult;
    HRESULT     hRes;
    char        szBuffer[128];

    switch (msg)
    {
        case WM_CREATE:
            hRes = CreateDirect3D(hwnd);
            if (FAILED(hRes))
            {
                ReportError(hwnd, IDS_ERRMSG_CREATEDEVICE, hRes);
                ReleaseDirect3D();
                return -1L;
            }

            hRes = CreatePrimary(hwnd);
            if (FAILED(hRes))
            {
                ReportError(hwnd, IDS_ERRMSG_INITSCREEN, hRes);
                ReleasePrimary();
                ReleaseDirect3D();
                return -1L;
            }

            hRes = ChooseDevice();
            if (FAILED(hRes))
            {
                ReportError(hwnd, IDS_ERRMSG_NODEVICE, hRes);
                ReleasePrimary();
                ReleaseDirect3D();
                return -1L;
            }

            // Update the title to show the name of the chosen device.

            wsprintf(szBuffer, "%s: %s", WINDOW_TITLE, szDeviceName);
            SetWindowText(hwnd, szBuffer);

            return 0L;

        case WM_MOVE:
            return OnMove(hwnd, (int)LOWORD(lParam),
                (int)HIWORD(lParam));

        case WM_SIZE:
            return OnSize(hwnd, (int)LOWORD(lParam),
                (int)HIWORD(lParam));
```

```
case WM_ERASEBKGND:
// Our rendering fills the entire viewport so we won't bother
// erasing the background.

    return 1L;

case WM_PAINT:
    hdc = BeginPaint(hwnd, &ps);

    lResult = OnPaint(hwnd, hdc, &ps);

    EndPaint(hwnd, &ps);
    return lResult;

case WM_ACTIVATEAPP:
    fActive = (BOOL)wParam;
    if (fActive && !fSuspended && (NULL != lpddPalette))
    {
        // Realizing the palette using DirectDraw is different
        // from GDI. To realize the palette we call SetPalette
        // each time our application is activated.

        // NOTE: DirectDraw recognizes that the new palette
        // is the same as the old one and so does not increase
        // the reference count of the palette.

        hRes = lpddPrimary->lpVtbl->SetPalette(lpddPrimary,
            lpddPalette);
        if (FAILED(hRes))
        {
            FatalError(hwnd, IDS_ERRMSG_REALIZEPALETTE, hRes);
            return 0L;
        }

    }
    else
    {
        // If we have been deactived, invalidate to show
        // the suspended display.

        InvalidateRect(hwnd, NULL, FALSE);
    }
    return 0L;

case WM_KEYUP:
    // We use the escape key as a quick way of
    // getting out of the application.

    if (VK_ESCAPE == (int)wParam)
    {
        DestroyWindow(hwnd);
        return 0L;
    }
    break;
```

```
        case WM_CLOSE:
            DestroyWindow(hwnd);
            return 0L;

        case WM_DESTROY:
            // All cleanup is done here when terminating normally or
            // shutting down due to an error.

            ReleaseScene();
            ReleaseDevice();
            ReleasePrimary();
            ReleaseDirect3D();

            PostQuitMessage(0);
            return 0L;
    }

    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

WinMain Function

```
int PASCAL
WinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR     lpszCommandLine,
        int       cmdShow)
{
    WNDCLASS wndClass;
    HWND     hwnd;
    MSG      msg;

    USE_PARAM(hPrevInstance);

    // Record the instance handle.

    hAppInstance = hInstance;

    // Very simple command-line processing. We only have one
    // option - debug - so we will just assume that if anything was
    // specified on the command line the user wants debug mode.
    // (In debug mode there is no hardware and all surfaces are
    // explicitly in system memory.)

    if (0 != *lpszCommandLine)
        fDebug = TRUE;

    // Register the window class.

    wndClass.style         = 0;
    wndClass.lpfnWndProc   = WndProc;
    wndClass.cbClsExtra    = 0;
    wndClass.cbWndExtra    = 0;
    wndClass.hInstance     = hInstance;
    wndClass.hIcon         = LoadIcon(hAppInstance,
        MAKEINTRESOURCE(IDI_APPICON));
    wndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
    wndClass.lpszMenuName  = NULL;
    wndClass.lpszClassName = WINDOW_CLASSNAME;

    RegisterClass(&wndClass);

    // Create the main window of the instance.

    hwnd = CreateWindow(WINDOW_CLASSNAME,
                        WINDOW_TITLE,
                        WS_OVERLAPPED | WS_SYSMENU,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        WINDOW_WIDTH, WINDOW_HEIGHT,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
```

```
    ShowWindow(hwnd, cmdShow);
    UpdateWindow(hwnd);

    // The main message dispatch loop.

    // NOTE: For simplicity we handle the message loop with a
    // simple PeekMessage scheme. This might not be the best
    // mechanism for a real application (a separate render worker
    // thread might be better).

    while (TRUE)
    {
        if (PeekMessage(&msg, NULL, 0U, 0U, PM_REMOVE))
        {
            // Message pending. If it's QUIT then exit the message
            // loop. Otherwise, process the message.

            if (WM_QUIT == msg.message)
            {
                break;
            }
            else
            {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else
        {
            // Animate the scene.

            OnIdle(hwnd);
        }
    }

    return msg.wParam;
}
```

Immediate-Mode Reference

This section contains reference information for the API elements provided by Direct3D® Immediate Mode. Reference material is divided into the following categories:

- **Interfaces**
- **D3D_OVERLOADS**
- **Callback Functions**
- **Macros**
- **Structures**
- **Enumerated Types**
- **Other Types**
- **Return Values**

Interfaces

This section contains reference information for the COM interfaces provided by Direct3D's Immediate Mode. The following interfaces are covered:

- **IDirect3D2**
- **IDirect3DDevice**
- **IDirect3DDevice2**
- **IDirect3DExecuteBuffer**
- **IDirect3DLight**
- **IDirect3DMaterial2**
- **IDirect3DTexture2**
- **IDirect3DViewport2**

IDirect3D2

Applications use the methods of the **IDirect3D2** interface to create Direct3D objects and set up the environment. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3D2 interface.

The **IDirect3D2** interface is obtained by calling the **QueryInterface** method from a DirectDraw object.

The major difference between **IDirect3D2** and the **IDirect3D** interface is the addition of the **CreateDevice** method.

The methods of the **IDirect3D2** interface can be organized into the following groups:

Creation	**CreateDevice**
	CreateLight
	CreateMaterial
	CreateViewport
Enumeration	**EnumDevices**
	FindDevice

The **IDirect3D2** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3D2** and **LPDIRECT3D** types are defined as pointers to the **IDirect3D2** and **IDirect3D** interfaces:

```
typedef struct IDirect3D    *LPDIRECT3D;
typedef struct IDirect3D2   *LPDIRECT3D2;
```

IDirect3D2::CreateDevice

The **IDirect3D2::CreateDevice** method creates a Direct3D device to be used with the DrawPrimitive methods.

```
HRESULT CreateDevice(
  REFCLSID rclsid,
  LPDIRECTDRAWSURFACE lpDDS,
  LPDIRECT3DDEVICE2 * lplpD3DDevice2
);
```

Parameters

rclsid
Class identifier for the new device. This can be IID_IDirect3DHALDevice, IID_IDirect3DMMXDevice, IID_IDirect3DRampDevice, or IID_IDirect3DRGBDevice.

lpDDS
Address of a DirectDraw surface that describes the new device.

lplpD3DDevice2
Address that points to the new **IDirect3DDevice2** interface when the method returns.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3D2** interface. In previous versions of Direct3D, devices could be created only by calling the **IDirectDrawSurface::QueryInterface** method; devices created in this manner can only be used with execute buffers.

When you call **IDirect3D2::CreateDevice**, you create a device object that is separate from a DirectDraw surface object. This device uses a DirectDraw surface as a rendering target.

IDirect3D2::CreateLight

The **IDirect3D2::CreateLight** method allocates a Direct3DLight object. This object can then be associated with a viewport by using the **IDirect3DViewport2::AddLight** method.

```
HRESULT CreateLight(
  LPDIRECT3DLIGHT* lplpDirect3DLight,
  IUnknown* pUnkOuter
);
```

Parameters

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 **IDirect3D2::CreateLight** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3D** interface.

IDirect3D2::CreateMaterial

The **IDirect3D2::CreateMaterial** method allocates a Direct3DMaterial2 object.

```
HRESULT CreateMaterial(
  LPDIRECT3DMATERIAL2* lplpDirect3DMaterial2,
  IUnknown* pUnkOuter
);
```

Parameters

lplpDirect3DMaterial2
Address that will be filled with a pointer to an **IDirect3DMaterial2** interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D2::CreateMaterial** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

Remarks

In the **IDirect3D** interface, this method retrieved a pointer to an **IDirect3DMaterial** interface, not an **IDirect3DMaterial2** interface.

IDirect3D2::CreateViewport

The **IDirect3D2::CreateViewport** method creates a Direct3DViewport object. The viewport is associated with a Direct3DDevice object by using the **IDirect3DDevice2::AddViewport** method.

```
HRESULT CreateViewport(
  LPDIRECT3DVIEWPORT2* lplpD3DViewport2,
  IUnknown* pUnkOuter
);
```

Parameters

lplpD3DViewport
Address that will be filled with a pointer to an **IDirect3DViewport2** interface if the call succeeds.

pUnkOuter
This parameter is provided for future compatibility with COM aggregation features. Currently, however, the **IDirect3D2::CreateViewport** method returns an error if this parameter is anything but NULL.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3D** interface, this method retrieves a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport2** interface.

IDirect3D2::EnumDevices

The **IDirect3D2::EnumDevices** method enumerates all Direct3D device drivers installed on the system.

```
HRESULT EnumDevices(
  LPD3DENUMDEVICESCALLBACK lpEnumDevicesCallback,
  LPVOID lpUserArg
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

MMX devices are enumerated only by **IDirect3D2::EnumDevices**, not by its predecessor, **IDirect3D::EnumDevices**. If you use the **QueryInterface** method to create an **IDirect3D** interface from **IDirect3D2** before you enumerate the Direct3D drivers, the enumeration will behave like **IDirect3D::EnumDevices** – no MMX devices will be enumerated.

To use execute buffers with an MMX device, you must call the **IDirect3D2::CreateDevice** method to create an MMX **IDirect3DDevice2** interface and then use the **QueryInterface** method to create an **IDirect3DDevice** interface from **IDirect3DDevice2**.

IDirect3D2::FindDevice

The **IDirect3D2::FindDevice** method finds a device with specified characteristics and retrieves a description of it.

```
HRESULT FindDevice(
  LPD3DFINDDEVICESEARCH lpD3DFDS,
  LPD3DFINDDEVICERESULT lpD3DFDR
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

Remarks

This method is unchanged from its implementation in the **IDirect3D** interface.

IDirect3D::Initialize

The **IDirect3D2::Initialize** method is not implemented.

```
HRESULT Initialize(
  REFIID lpREFIID
);
```

IDirect3DDevice

Applications use the methods of the **IDirect3DDevice** interface to retrieve and set the capabilities of Direct3D devices. This section is a reference to the methods of these interface. For a conceptual overview, see Devices.

The **IDirect3DDevice** interface supports applications that work with execute buffers. It has been extended by the **IDirect3DDevice2** interface, which supports the DrawPrimitive methods.

The Direct3DDevice object is obtained by calling the **QueryInterface** method from a DirectDrawSurface object that was created as a 3-D-capable surface.

The methods of the **IDirect3DDevice** interface can be organized into the following groups. Note that in some cases **IDirect3DDevice** methods are documented in the reference to the **IDirect3DDevice2** interface.

Execute buffers	**CreateExecuteBuffer**
	Execute
Information	**EnumTextureFormats**
	GetCaps
	GetDirect3D
	GetPickRecords
	GetStats
Matrices	**CreateMatrix**
	DeleteMatrix
	GetMatrix
	SetMatrix
Miscellaneous	**Initialize**
	Pick
	SwapTextureHandles
Scenes	**BeginScene**
	EndScene
Viewports	**AddViewport**
	DeleteViewport
	NextViewport

The **IDirect3DDevice** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DDEVICE** type is defined as a pointer to the **IDirect3DDevice** interface:

```
typedef struct IDirect3DDevice          *LPDIRECT3DDEVICE;
```

IDirect3DDevice::CreateExecuteBuffer

The **IDirect3DDevice::CreateExecuteBuffer** method allocates an execute buffer for a display list.

```
HRESULT CreateExecuteBuffer(
  LPD3DEXECUTEBUFFERDESC lpDesc,
  LPDIRECT3DEXECUTEBUFFER *lplpDirect3DExecuteBuffer,
  IUnknown *pUnkOuter
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The display 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.

The **D3DEXECUTEBUFFERDESC** structure describes the execute buffer to be created. At a minimum, the application must specify the size required. If the application specifies D3DDEBCAPS_VIDEOMEMORY in the **dwCaps** 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

The **IDirect3DDevice::CreateMatrix** method creates a matrix.

```
HRESULT CreateMatrix(
  LPD3DMATRIXHANDLE lpD3DMatHandle
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

See Also

IDirect3DDevice::DeleteMatrix, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::DeleteMatrix

The **IDirect3DDevice::DeleteMatrix** method deletes a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT DeleteMatrix(
  D3DMATRIXHANDLE d3dMatHandle
);
```

Parameters

d3dMatHandle
Matrix handle to be deleted.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::Execute

The **IDirect3DDevice::Execute** method executes a buffer.

```
HRESULT Execute(
  LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
  LPDIRECT3DVIEWPORT lpDirect3DViewport,
  DWORD dwFlags
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

D3DEXECUTEDATA, **D3DINSTRUCTION**, **IDirect3DExecuteBuffer::Validate**

IDirect3DDevice::GetMatrix

The **IDirect3DDevice::GetMatrix** method retrieves a matrix from a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT GetMatrix(
  D3DMATRIXHANDLE D3DMatHandle,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

D3DMatHandle
Handle to the matrix to be retrieved.

lpD3DMatrix
Address of a **D3DMATRIX** structure that contains the matrix when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::DeleteMatrix**, **IDirect3DDevice::SetMatrix**

IDirect3DDevice::GetPickRecords

The **IDirect3DDevice::GetPickRecords** method retrieves the pick records for a device.

```
HRESULT GetPickRecords(
  LPDWORD lpCount,
  LPD3DPICKRECORD lpD3DPickRec
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

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::Initialize

The **IDirect3DDevice::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3D lpd3d,
  LPGUID lpGUID,
  LPD3DDEVICEDESC lpd3ddvdesc
);
```

IDirect3DDevice::Pick

The **IDirect3DDevice::Pick** method executes a buffer without performing any rendering, but returns a z-ordered list of offsets to the primitives that intersect the upper-left corner of the rectangle specified by *lpRect*.

This call fails if the Direct3DExecuteBuffer object is locked.

```
HRESULT Pick(
  LPDIRECT3DEXECUTEBUFFER lpDirect3DExecuteBuffer,
  LPDIRECT3DVIEWPORT lpDirect3DViewport,
  DWORD dwFlags,
  LPD3DRECT lpRect
);
```

Parameters

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 device coordinates to be picked. Currently, only primitives that intersect the **x1, y1** coordinates of this rectangle are returned. The **x2, y2** coordinates are ignored.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

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.

See Also

IDirect3DDevice::GetPickRecords

IDirect3DDevice::SetMatrix

The **IDirect3DDevice::SetMatrix** method applies a matrix to a matrix handle. This matrix handle must have been created by using the **IDirect3DDevice::CreateMatrix** method.

```
HRESULT SetMatrix(
  D3DMATRIXHANDLE d3dMatHandle,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

d3dMatHandle
Matrix handle to be set.

lpD3DMatrix
Address of a **D3DMATRIX** structure that describes the matrix to be set.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error, such as DDERR_INVALIDPARAMS.

Remarks

Transformations inside the execute buffer include a handle to a matrix. The **IDirect3DDevice::SetMatrix** method enables an application to change this matrix without having to lock and unlock the execute buffer.

See Also

IDirect3DDevice::CreateMatrix, **IDirect3DDevice::GetMatrix**, **IDirect3DDevice::DeleteMatrix**

IDirect3DDevice2

The **IDirect3DDevice2** interface helps applications work with the DrawPrimitive methods; this is in contrast to the **IDirect3DDevice** interface, which applications use to work with execute buffers. You can create a Direct3DDevice2 object by calling the **IDirect3D2::CreateDevice** method.

For a conceptual overview, see Devices and The DrawPrimitive Methods.

The methods of the **IDirect3DDevice2** interface can be organized into the following groups:

Information	**EnumTextureFormats** **GetCaps** **GetDirect3D** **GetStats**
Miscellaneous	**MultiplyTransform** **SwapTextureHandles**
Getting and Setting States	**GetClipStatus** **GetCurrentViewport** **GetLightState** **GetRenderState** **GetRenderTarget** **GetTransform** **SetClipStatus** **SetCurrentViewport** **SetLightState** **SetRenderState** **SetRenderTarget** **SetTransform**
Rendering	**Begin** **BeginIndexed** **DrawIndexedPrimitive** **DrawPrimitive** **End** **Index** **Vertex**
Scenes	**BeginScene** **EndScene**
Viewports	**AddViewport** **DeleteViewport** **NextViewport**

The **IDirect3DDevice2** interface, like all COM interfaces, inherits the **IUnknown** interface methods.

The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **IDirect3DDevice2** interface is not intended to be used with execute buffers. If you need to use some of the methods in the **IDirect3DDevice** interface that are not supported in **IDirect3DDevice2**, you can call **IDirect3DDevice2::QueryInterface** to retrieve a pointer to an **IDirect3DDevice** interface. The following methods from the **IDirect3DDevice** interface are not supported by **IDirect3DDevice2**:

IDirect3DDevice::CreateExecuteBuffer
IDirect3DDevice::CreateMatrix
IDirect3DDevice::DeleteMatrix
IDirect3DDevice::Execute
IDirect3DDevice::GetMatrix
IDirect3DDevice::GetPickRecords
IDirect3DDevice::Initialize
IDirect3DDevice::Pick
IDirect3DDevice::SetMatrix

The **LPDIRECT3DDEVICE2** type is defined as a pointer to the **IDirect3DDevice2** interface:

```
typedef struct IDirect3DDevice2    *LPDIRECT3DDEVICE2;
```

IDirect3DDevice2::AddViewport

The **IDirect3DDevice2::AddViewport** method adds the specified viewport to the list of viewport objects associated with the device.

```
HRESULT AddViewport(
  LPDIRECT3DVIEWPORT2 lpDirect3DViewport2
);
```

Parameters

lpDirect3DViewport2
Address of the **IDirect3DViewport2** interface that should be associated with this Direct3DDevice object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DDevice** interface, this method requires a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport2** interface.

IDirect3DDevice2::Begin

The **IDirect3DDevice2::Begin** method indicates the start of a sequence of rendered primitives. This method defines the type of these primitives and the type of vertices on which they are based. The only method you can legally call between calls to **IDirect3DDevice2::Begin** and **IDirect3DDevice2::End** is **IDirect3DDevice2::Vertex**.

```
HRESULT Begin(
  D3DPRIMITIVETYPE d3dpt,
  D3DVERTEXTYPE d3dvt,
  DWORD dwFlags
);
```

Parameters

d3dpt

One of the members of the **D3DPRIMITIVETYPE** enumerated type.

d3dvt

Indicates the type of vertices to be used in rendering this primitive. Only vertices of this type will be accepted before the corresponding **IDirect3DDevice2::End**.

This must be one of the members of the **D3DVERTEXTYPE** enumerated type, as specified in a call to the **IDirect3DDevice2::Vertex** method.

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP	The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)
D3DDP_DONOTUPDATEEXTENTS	Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice2::GetClipStatus** will not have been updated to account for the data rendered by this call.
D3DDP_OUTOFORDER	A hint to the system that the primitives can be rendered out of order. Note that back-to-back calls to DrawPrimitive methods using this flag may cause triangles from the primitives to be interleaved. The DrawPrimitive methods that use this flag are **Begin**, **BeginIndexed**, **DrawIndexedPrimitive**, and **DrawPrimitive**.
D3DDP_WAIT	Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this

flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method fails if it is called after a call to the **IDirect3DDevice2::Begin** or **IDirect3DDevice2::BeginIndexed** method that has no bracketing call to **IDirect3DDevice2::End** method. Rendering calls that specify the wrong vertex type or that perform state changes will cause rendering of this primitive to fail.

This method was first introduced in the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::BeginIndexed, **IDirect3DDevice2::End**, **IDirect3DDevice2::Vertex**

IDirect3DDevice2::BeginIndexed

The **IDirect3DDevice2::BeginIndexed** method defines the start of a primitive based on indexing into an array of vertices. This method fails if it is called after a call to the **IDirect3DDevice2::Begin** or **IDirect3DDevice2::BeginIndexed** method that has no corresponding call to **IDirect3DDevice2::End**. The only method you can legally call between calls to **IDirect3DDevice2::BeginIndexed** and **IDirect3DDevice2::End** is **IDirect3DDevice2::Index**.

```
HRESULT BeginIndexed(
  D3DPRIMITIVETYPE dptPrimitiveType,
  D3DVERTEXTYPE dvtVertexType,
  LPVOID lpvVertices,
  DWORD dwNumVertices,
  DWORD dwFlags
);
```

Parameters

dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type. Note that the **D3DPT_POINTLIST** member of **D3DPRIMITIVETYPE** is not indexed.

dvtVertexType
Indicates the types of the vertices used. This must be one of the members of the **D3DVERTEXTYPE** enumerated type.

lpvVertices
Pointer to the list of vertices to be used in the primitive sequence.

dwNumVertices
Number of vertices in the above array.

dwFlags
One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP	The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)
D3DDP_DONOTUPDATEEXTENTS	Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice2::GetClipStatus** will not have been updated to account for the data rendered by this call.
D3DDP_OUTOFORDER	A hint to the system that the primitives can be rendered out of order. Note that back-to-back calls to DrawPrimitive methods using this flag may cause triangles from the primitives to be interleaved. The DrawPrimitive methods that use this flag are **Begin**, **BeginIndexed**, **DrawIndexedPrimitive**, and **DrawPrimitive**.
D3DDP_WAIT	Causes the method to wait until the

	polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was first introduced in the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::Begin, **IDirect3DDevice2::End**, **IDirect3DDevice2::Index**

IDirect3DDevice2::BeginScene

The **IDirect3DDevice2::BeginScene** method begins a scene.

Applications must call the **IDirect3DDevice2::BeginScene** method before performing any rendering, and must call **IDirect3DDevice2::EndScene** when rendering is complete.

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

```
HRESULT BeginScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

See Also

IDirect3DDevice2::EndScene

IDirect3DDevice2::DeleteViewport

The **IDirect3DDevice2::DeleteViewport** method removes the specified viewport from the list of viewport objects associated with the device.

```
HRESULT DeleteViewport(
  LPDIRECT3DVIEWPORT2 lpDirect3DViewport2
);
```

Parameters

lpDirect3DViewport2
Address of the Direct3DViewport2 object that should be disassociated with this Direct3DDevice2 object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DDevice** interface, this method requires a pointer to an **IDirect3DViewport** interface, not an **IDirect3DViewport2** interface.

IDirect3DDevice2::DrawIndexedPrimitive

The **IDirect3DDevice2::DrawIndexedPrimitive** method renders the specified geometric primitive based on indexing into an array of vertices.

```
HRESULT DrawIndexedPrimitive(
  D3DPRIMITIVETYPE d3dptPrimitiveType,
  D3DVERTEXTYPE d3dvtVertexType,
  LPVOID lpvVertices,
  DWORD dwVertexCount,
  LPWORD dwIndices,
  DWORD dwIndexCount,
  DWORD dwFlags
);
```

Parameters

d3dptPrimitiveType

Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

Note that the **D3DPT_POINTLIST** member of **D3DPRIMITIVETYPE** is not indexed.

d3dvtVertexType

Indicates the types of the vertices used. This must be one of the members of the **D3DVERTEXTYPE** enumerated type.

lpvVertices

Pointer to the list of vertices to be used in the primitive sequence.

dwVertexCount

Defines the number of vertices in the list.

Notice that this parameter is used differently from the *dwVertexCount* parameter in the **IDirect3DDevice2::DrawPrimitive** method. In that method, the *dwVertexCount* parameter gives the number of vertices to draw, but here it gives the total number of vertices in the array pointed to by the *lpvVertices* parameter. When you call **IDirect3DDevice2::DrawIndexedPrimitive**, you specify the number of vertices to draw in the *dwIndexCount* parameter.

dwIndices

Pointer to a list of WORDs that are to be used to index into the specified vertex list when creating the geometry to render.

dwIndexCount

Specifies the number of indices provided for creating the geometry.

dwFlags

One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP	The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)
D3DDP_DONOTUPDATEEXTENTS	Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice2::GetClipStatus** will not have been updated to account for the data rendered by this call.

D3DDP_OUTOFORDER	A hint to the system that the primitives can be rendered out of order. Note that back-to-back calls to DrawPrimitive methods using this flag may cause triangles from the primitives to be interleaved. The DrawPrimitive methods that use this flag are **Begin**, **BeginIndexed**, **DrawIndexedPrimitive**, and **DrawPrimitive**.
D3DDP_WAIT	Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

In current versions of DirectX, **IDirect3DDevice2::DrawIndexedPrimitive** can sometimes generate an update rectangle that is larger than it strictly needs to be. If a large number of vertices need to be processed, this can have a negative impact on the performance of your application. If you are using **D3DTLVERTEX** vertices and the system is processing more vertices than you need, you should use the D3DDP_DONOTCLIP and D3DDP_DONOTUPDATEEXTENTS flags to solve the problem.

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::DrawPrimitive

IDirect3DDevice2::DrawPrimitive

The **IDirect3DDevice2::DrawPrimitive** method renders the specified array of vertices as a sequence of geometric primitives of the specified type.

```
HRESULT DrawPrimitive(
  D3DPRIMITIVETYPE dptPrimitiveType,
  D3DVERTEXTYPE dvtVertexType,
  LPVOID lpvVertices,
  DWORD dwVertexCount,
  DWORD dwFlags
);
```

Parameters

dptPrimitiveType
Type of primitive to be rendered by this command. This must be one of the members of the **D3DPRIMITIVETYPE** enumerated type.

dvtVertexType
Indicates the types of the vertices used. This must be one of the members of the **D3DVERTEXTYPE** enumerated type.

lpvVertices
Pointer to the array of vertices to be used in the primitive sequence.

dwVertexCount
Defines the number of vertices in the array.

dwFlags
One or more of the following flags defining how the primitive is drawn:

D3DDP_DONOTCLIP	The application has already done the required clipping, so the system should not necessarily clip the primitives. (This flag is a hint; the system may clip the primitive even when this flag is specified, under some circumstances.)
D3DDP_DONOTUPDATEEXTENTS	Disables the updating of the screen rectangle affected by this rendering call. Using this flag can potentially help performance, but the extents returned by **IDirect3DDevice2::GetClipStatus** will not have been updated to account for the data rendered by this call.
D3DDP_OUTOFORDER	A hint to the system that the primitives can be rendered out of order. Note that back-to-back calls to DrawPrimitive methods using this flag may cause triangles from the primitives to be interleaved. The DrawPrimitive methods that use this flag are **Begin**, **BeginIndexed**, **DrawIndexedPrimitive**, and **DrawPrimitive**.
D3DDP_WAIT	Causes the method to wait until the polygons have been rendered before it returns, instead of returning as soon as the polygons have been sent to the card. (On scene-capture cards, the method

	returns as soon as the card responds.) This flag is typically used for debugging. Applications should not attempt to use this flag to ensure that a scene is up-to-date before continuing.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::DrawIndexedPrimitive

IDirect3DDevice2::End

The **IDirect3DDevice2::End** method signals the completion of a primitive sequence. This method fails if no corresponding call to the **IDirect3DDevice2::Begin** method was made.

```
HRESULT End(
  DWORD dwFlags
);
```

Parameters

dwFlags
Reserved. A flag word that should be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method fails if the vertex count is incorrect for the primitive type. It fails without drawing if it is called before a sufficient number of vertices is specified. If the number of Vertex or index calls made is not evenly divisible by 3 (in the case of triangles), or 2 (in the case of lineList), the remainder will be ignored.

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::Begin

IDirect3DDevice2::EndScene

The **IDirect3DDevice2::EndScene** method ends a scene that was begun by calling the **IDirect3DDevice2::BeginScene** method.

```
HRESULT EndScene();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

When this method succeeds, the scene will have been rendered and the device surface will hold the contents of the rendering.

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

See Also

IDirect3DDevice2::BeginScene

IDirect3DDevice2::EnumTextureFormats

The **IDirect3DDevice2::EnumTextureFormats** method queries the current driver for a list of supported texture formats.

```
HRESULT EnumTextureFormats(
  LPD3DENUMTEXTUREFORMATSCALLBACK lpd3dEnumTextureProc,
  LPVOID lpArg
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

IDirect3DDevice2::GetCaps

The **IDirect3DDevice2::GetCaps** method retrieves the capabilities of the Direct3DDevice2 object.

```
HRESULT GetCaps(
  LPD3DDEVICEDESC lpD3DHWDevDesc,
  LPD3DDEVICEDESC lpD3DHELDevDesc
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method does not retrieve the capabilities of the display device. To retrieve this information, use the **IDirectDraw2::GetCaps** method.

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

IDirect3DDevice2::GetClipStatus

The **IDirect3DDevice2::GetClipStatus** method gets the current clip status.

```
HRESULT GetClipStatus(
  LPD3DCLIPSTATUS lpD3DClipStatus
);
```

Parameters

lpD3DClipStatus
Address of a **D3DCLIPSTATUS** structure that describes the current clip status.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetClipStatus

IDirect3DDevice2::GetCurrentViewport

The **IDirect3DDevice2::GetCurrentViewport** method retrieves the current viewport.

```
HRESULT GetCurrentViewport(
  LPDIRECT3DVIEWPORT2 *lplpd3dViewport2
);
```

Parameters

lplpd3dViewport2

Address that contains a pointer to the current viewport when the method returns. A reference is taken to the viewport object.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
D3DERR_NOCURRENTVIEWPORT	No current viewport has been set by a call to the **IDirect3DDevice2::SetCurrentViewport** method.

Remarks

This method increases the reference count of the viewport interface retrieved in the *lplpd3dViewport2* parameter. The application must release this interface when it is no longer needed.

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetCurrentViewport

IDirect3DDevice2::GetDirect3D

The **IDirect3DDevice2::GetDirect3D** method retrieves the current **IDirect3D2** interface.

```
HRESULT GetDirect3D(
  LPDIRECT3D2 *lplpD3D2
);
```

Parameters

lplpD3D2
Address that will contain the interface when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return codes, see Direct3D Immediate-Mode Return Values.

Remarks

In the **IDirect3DDevice** interface, this method retrieves the current **IDirect3D** interface instead of an **IDirect3D2** interface.

IDirect3DDevice2::GetLightState

The **IDirect3DDevice2::GetLightState** method gets a single Direct3D Device lighting-related state value.

```
HRESULT GetLightState(
  D3DLIGHTSTATETYPE dwLightStateType,
  LPDWORD lpdwLightState
);
```

Parameters

dwLightStateType
Device state variable that is being queried. This parameter can be any of the members of the **D3DLIGHTSTATETYPE** enumerated type.

lpdwLightState
Address of a variable that will contain the Direct3D Device light state when the method returns.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetLightState

IDirect3DDevice2::GetRenderState

The **IDirect3DDevice2::GetRenderState** method gets a single Direct3D Device rendering state parameter.

```
HRESULT GetRenderState(
  D3DRENDERSTATETYPE dwRenderStateType,
  LPDWORD lpdwRenderState
);
```

Parameters

dwRenderStateType
Device state variable that is being queried. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

lpdwRenderState
Address of a variable that will contain the Direct3D Device render state when the method returns.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetRenderState

IDirect3DDevice2::GetRenderTarget

The **IDirect3DDevice2::GetRenderTarget** method retrieves a pointer to the DirectDraw surface that is being used as a render target.

```
HRESULT GetRenderTarget(
  LPDIRECTDRAWSURFACE *lplpRenderTarget
);
```

Parameters

lplpRenderTarget
Address that will contain a pointer to the DirectDraw surface object that is being used as a render target by this Direct3D device.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetRenderTarget

IDirect3DDevice2::GetStats

The **IDirect3DDevice2::GetStats** method retrieves statistics about a device.

```
HRESULT GetStats(
  LPD3DSTATS lpD3DStats
);
```

Parameters

lpD3DStats

Address of a **D3DSTATS** structure that will be filled with the statistics.

Return Values

If the method succeeds, the return value isD3D_OK .

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DDevice** interface.

IDirect3DDevice2::GetTransform

The **IDirect3DDevice2::GetTransform** method gets a matrix describing a transformation state.

```
HRESULT GetTransform(
  D3DTRANSFORMSTATETYPE dtstTransformStateType,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

dtstTransformStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix
Address of a **D3DMATRIX** structure describing the transformation.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::SetTransform

IDirect3DDevice2::Index

The **IDirect3DDevice2::Index** method adds a new index to the currently started primitive.

```
HRESULT Index(
  WORD wVertexIndex
);
```

Parameters

wVertexIndex

Index of the next vertex to be added to the currently started primitive sequence.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

IDirect3DDevice2::MultiplyTransform

The **IDirect3DDevice2::MultiplyTransform** method modifies the current world matrix by combining it with a specified matrix. The multiplication order is *lpD3DMatrix* times *dtstTransformStateType*.

```
HRESULT MultiplyTransform(
  D3DTRANSFORMSTATETYPE dtstTransformStateType,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

dtstTransformStateType
One of the members of the **D3DTRANSFORMSTATETYPE** enumerated type. Only the D3DTRANSFORMSTATE_WORLD setting is likely to be useful. The matrix referred to by this parameter is modified by this method.

lpD3DMatrix
Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

An application might use the **IDirect3DDevice2::MultiplyTransform** method to work with hierarchies of transformations. For example, the geometry and transformations describing an arm might be arranged in the following hierarchy:

```
shoulder_transformation
    upper_arm geometry
    elbow transformation
        lower_arm geometry
        wrist transformation
            hand geometry
```

An application might use the following series of calls to render this hierarchy. (Not all of the parameters are shown in this pseudocode.)

```
IDirect3DDevice2::SetTransform(D3DTRANSFORMSTATE_WORLD,
    shoulder_transform)
IDirect3DDevice2::DrawPrimitive(upper_arm)
IDirect3DDevice2::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    elbow_transform)
IDirect3DDevice2::DrawPrimitive(lower_arm)
IDirect3DDevice2::MultiplyTransform(D3DTRANSFORMSTATE_WORLD,
    wrist_transform)
IDirect3DDevice2::DrawPrimitive(hand)
```

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::DrawPrimitive, **IDirect3DDevice2::SetTransform**

IDirect3DDevice2::NextViewport

The **IDirect3DDevice2::NextViewport** method enumerates the viewports associated with the device.

```
HRESULT NextViewport(
  LPDIRECT3DVIEWPORT2 lpDirect3DViewport2,
  LPDIRECT3DVIEWPORT2 *lplpDirect3DViewport2,
  DWORD dwFlags
);
```

Parameters

lpDirect3DViewport2
Address of a viewport in the list of viewports associated with this Direct3DDevice2 object.

lplpDirect3DViewport2
Address of the next viewport in the list of viewports associated with this Direct3DDevice2 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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

If you attempt to retrieve the next viewport in the list when you are at the end of the list, this method returns D3D_OK but *lplpDirect3DViewport2* is NULL.

In the **IDirect3DDevice** interface, this method requires pointers to **IDirect3DViewport** interfaces, not **IDirect3DViewport2** interfaces.

IDirect3DDevice2::SetClipStatus

The **IDirect3DDevice2::SetClipStatus** method sets the current clip status.

```
HRESULT SetClipStatus(
  LPD3DCLIPSTATUS lpD3DClipStatus
);
```

Parameters

lpD3DClipStatus
Address of a **D3DCLIPSTATUS** structure that describes the new settings for the clip status.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetClipStatus

IDirect3DDevice2::SetCurrentViewport

The **IDirect3DDevice2::SetCurrentViewport** method sets the current viewport.

```
HRESULT SetCurrentViewport(
  LPDIRECT3DVIEWPORT2 lpd3dViewport2
);
```

Parameters

lpd3dViewport2
Address of the viewport that will become the current viewport if the method is successful.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

Applications must call this method before calling any rendering functions. Before calling this method, applications must have already called the **IDirect3DDevice2::AddViewport** method to add the viewport to the device.

Before the first call to **IDirect3DDevice2::SetCurrentViewport**, the current viewport for the device is invalid, and any attempts to render using the device will fail.

This method increases the reference count of the viewport interface specified by the *lpd3dViewport2* parameter and releases the previous viewport, if any.

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetCurrentViewport

IDirect3DDevice2::SetLightState

The **IDirect3DDevice2::SetLightState** method sets a single Direct3D Device lighting-related state value.

```
HRESULT SetLightState(
  D3DLIGHTSTATETYPE dwLightStateType,
  DWORD dwLightState
);
```

Parameters

dwLightStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DLIGHTSTATETYPE** enumerated type.

dwLightState
New value for the Direct3D Device light state. The meaning of this parameter is dependent on the value specified for *dwLightStateType*. For example, if *dwLightStateType* were **D3DLIGHTSTATE_COLORMODEL**, the second parameter would be one of the members of the **D3DCOLORMODEL** enumerated type.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetLightState, **IDirect3DDevice2::SetRenderState**, **IDirect3DDevice2::SetTransform**

IDirect3DDevice2::SetRenderState

The **IDirect3DDevice2::SetRenderState** method sets a single Direct3D Device rendering state parameter.

```
HRESULT SetRenderState(
  D3DRENDERSTATETYPE dwRenderStateType,
  DWORD dwRenderState
);
```

Parameters

dwRenderStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DRENDERSTATETYPE** enumerated type.

dwRenderState
New value for the Direct3D Device render state. The meaning of this parameter is dependent on the value specified for *dwRenderStateType*. For example, if *dwRenderStateType* were **D3DRENDERSTATE_SHADEMODE**, the second parameter would be one of the members of the **D3DSHADEMODE** enumerated type.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetRenderState, **IDirect3DDevice2::SetLightState**, **IDirect3DDevice2::SetTransform**

IDirect3DDevice2::SetRenderTarget

The **IDirect3DDevice2::SetRenderTarget** method permits the application to easily route rendering output to a new DirectDraw surface as a render target.

```
HRESULT SetRenderTarget(
  LPDIRECTDRAWSURFACE lpNewRenderTarget,
  DWORD dwFlags
);
```

Parameters

lpNewRenderTarget
Pointer to the previously created DirectDraw surface object to which future rendering on this Direct3D Device will be directed.

dwFlags
A flag word that should be set to 0.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The error may be one of the following values:

DDERR_INVALIDPARAMS	One of the arguments is invalid.
DDERR_INVALIDSURFACETYPE	The surface passed as the first parameter is invalid.

Remarks

When you change the rendering target, all of the handles associated with the previous rendering target become invalid. This means that you will have to reacquire all of the texture handles. If you are using ramp mode, you should also update the texture handles inside materials, by calling the **IDirect3DMaterial2::SetMaterial** method. Any execute buffers (which have embedded handles) also need to be updated. The **IDirect3DDevice2::SetRenderState** method is most useful to applications that use the DrawPrimitive methods, especially when these applications do not use ramp mode.

If the new render target surface has different dimensions from the old (length, width, pixel-format), this method marks the viewport as invalid. The viewport may be revalidated after calling **IDirect3DDevice2::SetRenderTarget** by calling **IDirect3DViewport2::SetViewport** to restate viewport parameters that are compatible with the new surface.

Capabilities do not change with changes in the properties of the render target surface. Both the Direct3D HAL and the software rasterizers have only one opportunity to expose capabilities to the application. The system cannot expose different sets of capabilities depending on the format of the destination surface.

If a z-buffer is attached to the new render target, it replaces the previous z-buffer for the context. Otherwise, the old z-buffer is detached and z-buffering is disabled.

If more than one z-buffer is attached to the render target, this function fails.

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetRenderTarget

IDirect3DDevice2::SetTransform

The **IDirect3DDevice2::SetTransform** method sets a single Direct3D Device transformation-related state.

```
HRESULT SetTransform(
  D3DTRANSFORMSTATETYPE dtstTransformStateType,
  LPD3DMATRIX lpD3DMatrix
);
```

Parameters

dtstTransformStateType
Device state variable that is being modified. This parameter can be any of the members of the **D3DTRANSFORMSTATETYPE** enumerated type.

lpD3DMatrix
Address of a **D3DMATRIX** structure that modifies the current transformation.

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value is an error. The method returns DDERR_INVALIDPARAMS if one of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

See Also

IDirect3DDevice2::GetTransform, **IDirect3DDevice2::SetLightState**, **IDirect3DDevice2::SetRenderState**

IDirect3DDevice2::SwapTextureHandles

The **IDirect3DDevice2::SwapTextureHandles** method swaps two texture handles.

```
HRESULT SwapTextureHandles(
  LPDIRECT3DTEXTURE2 lpD3DTex1,
  LPDIRECT3DTEXTURE2 lpD3DTex2
);
```

Parameters

lpD3DTex1 and *lpD3DTex2*
Addresses of the textures whose handles will be swapped when the method returns.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error.

Remarks

This method is useful when an application is changing all the textures in a complicated object.

In the **IDirect3DDevice** interface, this method requires pointers to **IDirect3DTexture** interfaces, not **IDirect3DTexture2** interfaces.

IDirect3DDevice2::Vertex

The **IDirect3DDevice2::Vertex** method adds a new Direct3D vertex to the currently started primitive.

```
HRESULT Vertex(
  LPVOID lpVertexType
);
```

Parameters

lpVertexType

Pointer to the next Direct3D vertex to be added to the currently started primitive sequence. This can be any of the Direct3D vertex types: **D3DLVERTEX**, **D3DTLVERTEX**, or **D3DVERTEX**

Return Values

If the method succeeds, the return value is DD_OK.

If the method fails, the return value may be one of the following values:

D3DERR_INVALIDRAMPTEXTURE	Ramp mode is being used and the texture handle in the current material does not match the current texture handle that is set as a render state.
DDERR_INVALIDPARAMS	One of the arguments is invalid.

Remarks

This method was introduced with the **IDirect3DDevice2** interface.

IDirect3DExecuteBuffer

Applications use the methods of the **IDirect3DExecuteBuffer** interface to set up and control Direct3D execute buffers. This section is a reference to the methods of this interface. For a conceptual overview, see Execute Buffers.

The methods of the **IDirect3DExecuteBuffer** interface can be organized into the following groups:

Execute data	**GetExecuteData** **SetExecuteData**
Lock and unlock	**Lock** **Unlock**
Miscellaneous	**Initialize** **Optimize** **Validate**

The **IDirect3DExecuteBuffer** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DEXECUTEBUFFER** type is defined as a pointer to the **IDirect3DExecuteBuffer** interface:

```
typedef struct IDirect3DExecuteBuffer    *LPDIRECT3DEXECUTEBUFFER;
```

IDirect3DExecuteBuffer::GetExecuteData

The **IDirect3DExecuteBuffer::GetExecuteData** method retrieves the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

```
HRESULT GetExecuteData(
  LPD3DEXECUTEDATA lpData
);
```

Parameters

lpData

Address of a **D3DEXECUTEDATA** structure that will be filled with the current execute data state of the Direct3DExecuteBuffer object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This call fails if the Direct3DExecuteBuffer object is locked.

See Also

IDirect3DExecuteBuffer::SetExecuteData

IDirect3DExecuteBuffer::Initialize

The **IDirect3DExecuteBuffer::Initialize** method is provided for compliance with the COM protocol.

```
HRESULT Initialize(
  LPDIRECT3DDEVICE lpDirect3DDevice,
  LPD3DEXECUTEBUFFERDESC lpDesc
);
```

Parameters

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.

Return Values

The method returns DDERR_ALREADYINITIALIZED because the Direct3DExecuteBuffer object is initialized when it is created.

IDirect3DExecuteBuffer::Lock

The **IDirect3DExecuteBuffer::Lock** method obtains a direct pointer to the commands in the execute buffer.

```
HRESULT Lock(
  LPD3DEXECUTEBUFFERDESC lpDesc
);
```

Parameters

lpDesc

Address of a **D3DEXECUTEBUFFERDESC** structure. When the method returns, the **lpData** member will be set to point to the actual data to which the application has access. 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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_WASSTILLDRAWING

Remarks

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.

See Also

IDirect3DExecuteBuffer::Unlock

IDirect3DExecuteBuffer::Optimize

The **IDirect3DExecuteBuffer::Optimize** method is not currently supported.

```
HRESULT Optimize();
```

IDirect3DExecuteBuffer::SetExecuteData

The **IDirect3DExecuteBuffer::SetExecuteData** method sets the execute data state of the Direct3DExecuteBuffer object. The execute data is used to describe the contents of the Direct3DExecuteBuffer object.

```
HRESULT SetExecuteData(
  LPD3DEXECUTEDATA lpData
);
```

Parameters

lpData
Address of a **D3DEXECUTEDATA** structure that describes the execute buffer layout.

Return Values

If the method succeeds, the return value is D3D_OK .

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_LOCKED
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This call fails if the Direct3DExecuteBuffer object is locked.

See Also

IDirect3DExecuteBuffer::GetExecuteData

IDirect3DExecuteBuffer::Unlock

The **IDirect3DExecuteBuffer::Unlock** method 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.

```
HRESULT Unlock();
```

Parameters

None.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

D3DERR_EXECUTE_NOT_LOCKED
DDERR_INVALIDOBJECT

See Also

IDirect3DExecuteBuffer::Lock

IDirect3DExecuteBuffer::Validate

The **IDirect3DExecuteBuffer::Validate** method is not currently implemented.

```
HRESULT Validate(
  LPDWORD lpdwOffset,
  LPD3DVALIDATECALLBACK lpFunc,
  LPVOID lpUserArg,
  DWORD dwReserved
);
```

IDirect3DLight

Applications use the methods of the **IDirect3DLight** interface to retrieve and set the capabilities of lights. This section is a reference to the methods of this interface. For a conceptual overview, see Lights.

The **IDirect3DLight** interface is obtained by calling the **IDirect3D2::CreateLight** method.

The methods of the **IDirect3DLight** interface can be organized into the following groups:

Get and set	**GetLight**
	SetLight
Initialization	**Initialize**

The **IDirect3DLight** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DLIGHT** type is defined as a pointer to the **IDirect3DLight** interface:

```
typedef struct IDirect3DLight    *LPDIRECT3DLIGHT;
```

IDirect3DLight::GetLight

The **IDirect3DLight::GetLight** method retrieves the light information for the Direct3DLight object.

```
HRESULT GetLight(
  LPD3DLIGHT lpLight
);
```

Parameters

lpLight
Address of a **D3DLIGHT2** structure that will be filled with the current light data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirect3DLight::SetLight

IDirect3DLight::Initialize

The **IDirect3DLight::Initialize** method is provided for compliance with the COM protocol.

```
HRESULT Initialize(
  LPDIRECT3D lpDirect3D
);
```

Parameters

lpDirect3D

Address of the Direct3D structure representing the Direct3D object.

Return Values

The method returns DDERR_ALREADYINITIALIZED because the Direct3DLight object is initialized when it is created.

IDirect3DLight::SetLight

The **IDirect3DLight::SetLight** method sets the light information for the Direct3DLight object.

```
HRESULT SetLight(
  LPD3DLIGHT lpLight
);
```

Parameters

lpLight
Address of a **D3DLIGHT2** structure that will be used to set the current light data.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirect3DLight::GetLight

IDirect3DMaterial2

Applications use the methods of the **IDirect3DMaterial2** interface to retrieve and set the properties of materials. This section is a reference to the methods of this interface. For a conceptual overview, see Materials.

The **IDirect3DMaterial2** interface is an extension of the **IDirect3DMaterial** interface. You create this interface by calling the **IDirect3D2::CreateMaterial** method.

The methods of the **IDirect3DMaterial2** interface can be organized into the following groups:

Handles	**GetHandle**
Materials	**GetMaterial**
	SetMaterial

The **IDirect3DMaterial2** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DMATERIAL2** and **LPDIRECT3DMATERIAL** types are defined as pointers to the **IDirect3DMaterial2** and **IDirect3DMaterial** interfaces:

```
typedef struct IDirect3DMaterial2    *LPDIRECT3DMATERIAL2;
typedef struct IDirect3DMaterial     *LPDIRECT3DMATERIAL;
```

IDirect3DMaterial2::GetHandle

The **IDirect3DMaterial2::GetHandle** method obtains the material handle of the Direct3DMaterial object. This handle is used in all Direct3D methods in which 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.

```
HRESULT GetHandle(
  LPDIRECT3DDEVICE2 lpDirect3DDevice2,
  LPD3DMATERIALHANDLE lpHandle
);
```

Parameters

lpDirect3DDevice2
Address of the Direct3DDevice2 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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is DDERR_INVALIDOBJECT.

Remarks

In the **IDirect3DMaterial** interface, this method uses a pointer to a Direct3DMaterial object instead of a Direct3DMaterial2 object.

IDirect3DMaterial2::GetMaterial

The **IDirect3DMaterial2::GetMaterial** method retrieves the material data for the Direct3DMaterial object.

```
HRESULT GetMaterial(
  LPD3DMATERIAL lpMat
);
```

Parameters

lpMat
Address of a **D3DMATERIAL** structure that will be filled with the current material properties.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DMaterial** interface.

See Also

IDirect3DMaterial2::SetMaterial

IDirect3DMaterial::Initialize

The **IDirect3DMaterial2::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3D lpDirect3D
);
```

IDirect3DMaterial::Reserve

The **IDirect3DMaterial2::Reserve** method is not implemented.

```
HRESULT Reserve();
```

IDirect3DMaterial2::SetMaterial

The **IDirect3DMaterial2::SetMaterial** method sets the material data for the Direct3DMaterial object.

```
HRESULT SetMaterial(
  LPD3DMATERIAL lpMat
);
```

Parameters

lpMat
Address of a **D3DMATERIAL** structure that contains the material properties.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DMaterial** interface.

See Also

IDirect3DMaterial2::GetMaterial

IDirect3DMaterial::Unreserve

The **IDirect3DMaterial2::Unreserve** method is not implemented.

```
HRESULT Unreserve();
```

IDirect3DTexture2

Applications use the methods of the **IDirect3DTexture2** interface to retrieve and set the properties of textures. This section is a reference to the methods of this interface. For a conceptual overview, see Textures.

The **IDirect3DTexture2** interface is an extension of the **IDirect3DTexture** interface. You create this interface by calling the **IDirectDrawSurface::QueryInterface** method from the DirectDrawSurface object that was created as a texture map.

The methods of the **IDirect3DTexture2** interface can be organized into the following groups:

Handles	**GetHandle**
Loading	**Load**
Palette information	**PaletteChanged**

The **IDirect3DTexture2** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DTEXTURE2** and **LPDIRECT3DTEXTURE** types are defined as pointers to the **IDirect3DTexture2** and **IDirect3DTexture** interfaces:

```
typedef struct IDirect3DTexture2    *LPDIRECT3DTEXTURE2;
typedef struct IDirect3DTexture     *LPDIRECT3DTEXTURE;
```

IDirect3DTexture2::GetHandle

The **IDirect3DTexture2::GetHandle** method obtains the texture handle for the Direct3DTexture2 object. This handle is used in all Direct3D methods in which a texture is to be referenced.

```
HRESULT GetHandle(
  LPDIRECT3DDEVICE2 lpDirect3DDevice2,
  LPD3DTEXTUREHANDLE lpHandle
);
```

Parameters

lpDirect3DDevice2
Address of the Direct3DDevice2 object into which the texture is to be loaded.

lpHandle
Address that will contain the texture handle corresponding to the Direct3DTexture2 object.

Return Values

If the method succeeds, the return value is D3D_OK .

If the method fails, the return value may be one of the following values:

DDERR_INVALIDPARAMS

Remarks

In the **IDirect3DTexture** interface, this method uses a pointer to a Direct3DDevice object instead of a Direct3DDevice2 object.

IDirect3DTexture::Initialize

The **IDirect3DTexture2::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3DDEVICE lpD3DDevice,
  LPDIRECTDRAWSURFACE lpDDSurface
);
```

IDirect3DTexture2::Load

The **IDirect3DTexture2::Load** method loads a texture that was created with the DDSCAPS_ALLOCONLOAD flag, which indicates that memory for the DirectDraw surface is not allocated until this method loads the surface.

```
HRESULT Load(
  LPDIRECT3DTEXTURE2 lpD3DTexture2
);
```

Parameters

lpD3DTexture2
Address of the texture to load.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

Remarks

In the **IDirect3DTexture** interface, this method uses a pointer to a Direct3DTexture object instead of a Direct3DTexture2 object.

See Also

IDirect3DTexture::Unload

IDirect3DTexture2::PaletteChanged

The **IDirect3DTexture2::PaletteChanged** method informs the driver that the palette has changed on a surface.

```
HRESULT PaletteChanged(
  DWORD dwStart,
  DWORD dwCount
);
```

Parameters

dwStart
Index of first palette entry that has changed.

dwCount
Number of palette entries that have changed.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value is an error. For a list of possible return values, see Direct3D Immediate-Mode Return Values.

Remarks

This method is particularly useful for applications that play video clips and therefore require palette-changing capabilities.

This method is unchanged from its implementation in the **IDirect3DTexture** interface.

IDirect3DTexture::Unload

The **IDirect3DTexture2::Unload** method is not implemented.

```
HRESULT Unload();
```

IDirect3DViewport2

Applications use the methods of the **IDirect3DViewport2** interface to retrieve and set the properties of viewports. This section is a reference to the methods of this interface. For a conceptual overview, see Viewports and Transformations.

The **IDirect3DViewport2** interface is an extension of the **IDirect3DViewport** interface. You create the **IDirect3DViewport2** interface by calling the **IDirect3D2::CreateViewport** method.

The methods of the **IDirect3DViewport2** interface can be organized into the following groups:

Backgrounds	**GetBackground** **GetBackgroundDepth** **SetBackground** **SetBackgroundDepth**
Lights	**AddLight** **DeleteLight** **LightElements** **NextLight**
Materials and viewports	**Clear** **GetViewport** **GetViewport2** **SetViewport** **SetViewport2**
Transformation	**TransformVertices**

IDirect3DViewport2 is identical to **IDirect3DViewport** except for two new methods: **GetViewport2** and **SetViewport2**. The **IDirect3DViewport2** interface differs from the **IDirect3DViewport** interface primarily in its use of the **D3DVIEWPORT2** structure. This structure introduces a closer correspondence between window size and viewport size than is true for the **D3DVIEWPORT** structure.

The **IDirect3DViewport** interface, like all COM interfaces, inherits the **IUnknown** interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

The **LPDIRECT3DVIEWPORT2** and **LPDIRECT3DVIEWPORT** types are defined as pointers to the **IDirect3DViewport2** and **IDirect3DViewport** interfaces:

```
typedef struct IDirect3DViewport2    *LPDIRECT3DVIEWPORT2;
typedef struct IDirect3DViewport     *LPDIRECT3DVIEWPORT;
```

IDirect3DViewport2::AddLight

The **IDirect3DViewport2::AddLight** method adds the specified light to the list of Direct3DLight objects associated with this viewport.

```
HRESULT AddLight(
  LPDIRECT3DLIGHT lpDirect3DLight
);
```

Parameters

lpDirect3DLight
Address of the Direct3DLight object that should be associated with this Direct3DDevice object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

IDirect3DViewport2::Clear

The **IDirect3DViewport2::Clear** method clears the viewport or a set of rectangles in the viewport to the current background material.

```
HRESULT Clear(
  DWORD dwCount,
  LPD3DRECT lpRects,
  DWORD dwFlags
);
```

Parameters

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).

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

IDirect3DViewport2::DeleteLight

The **IDirect3DViewport2::DeleteLight** method removes the specified light from the list of Direct3DLight objects associated with this viewport.

```
HRESULT DeleteLight(
  LPDIRECT3DLIGHT lpDirect3DLight
);
```

Parameters

lpDirect3DLight
Address of the Direct3DLight object that should be disassociated with this Direct3DDevice object.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

IDirect3DViewport2::GetBackground

The **IDirect3DViewport2::GetBackground** method retrieves the handle to a material that represents the current background associated with the viewport.

```
HRESULT GetBackground(
  LPD3DMATERIALHANDLE lphMat,
  LPBOOL lpValid
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::SetBackground

IDirect3DViewport2::GetBackgroundDepth

The **IDirect3DViewport2::GetBackgroundDepth** method retrieves a DirectDraw surface that represents the current background-depth field associated with the viewport.

```
HRESULT GetBackgroundDepth(
  LPDIRECTDRAWSURFACE* lplpDDSurface,
  LPBOOL lpValid
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::SetBackgroundDepth

IDirect3DViewport2::GetViewport

The **IDirect3DViewport2::GetViewport** method retrieves the viewport registers of the viewport. In the **IDirect3DViewport2** interface, this method has been superseded by the **IDirect3DViewport2::GetViewport2** method.

```
HRESULT GetViewport(
  LPD3DVIEWPORT lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT** structure representing the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT_dx5_DDERR_INVALIDOBJECT_ddraw
DDERR_INVALIDPARAMS_dx5_DDERR_INVALIDPARAMS_ddraw

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::GetViewport2, **IDirect3DViewport2::SetViewport**

IDirect3DViewport2::GetViewport2

The **IDirect3DViewport2::GetViewport2** method retrieves the viewport registers of the viewport.

```
HRESULT GetViewport2(
  LPD3DVIEWPORT2 lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT2** structure representing the viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirect3DViewport2::SetViewport2

IDirect3DViewport2::Initialize

The **IDirect3DViewport2::Initialize** method is not implemented.

```
HRESULT Initialize(
  LPDIRECT3D lpDirect3D
);
```

IDirect3DViewport2::LightElements

The **IDirect3DViewport2::LightElements** method is not currently implemented.

```
HRESULT LightElements(
  DWORD dwElementCount,
  LPD3DLIGHTDATA lpData
);
```

IDirect3DViewport2::NextLight

The **IDirect3DViewport2::NextLight** method enumerates the Direct3DLight objects associated with the viewport.

```
HRESULT NextLight(
  LPDIRECT3DLIGHT lpDirect3DLight,
  LPDIRECT3DLIGHT* lplpDirect3DLight,
  DWORD dwFlags
);
```

Parameters

lpDirect3DLight
Address of a light in the list of lights associated with this viewport object.

lplpDirect3DLight
Address of a pointer that will contain the requested light in the list of lights associated with this viewport 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.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

IDirect3DViewport2::SetBackground

The **IDirect3DViewport2::SetBackground** method sets the background associated with the viewport.

```
HRESULT SetBackground(
  D3DMATERIALHANDLE hMat
);
```

Parameters

hMat
Material handle that will be used as the background.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::GetBackground

IDirect3DViewport2::SetBackgroundDepth

The **IDirect3DViewport2::SetBackgroundDepth** method sets the background-depth field for the viewport.

```
HRESULT SetBackgroundDepth(
  LPDIRECTDRAWSURFACE lpDDSurface
);
```

Parameters

lpDDSurface
Address of the DirectDrawSurface object representing the background depth.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

The z-buffer is filled with the specified depth field when the **IDirect3DViewport2::Clear** method is called and the D3DCLEAR_ZBUFFER flag is specified. The bit depth must be 16 bits.

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::GetBackgroundDepth

IDirect3DViewport2::SetViewport

The **IDirect3DViewport2::SetViewport** method sets the viewport registers of the viewport. In the **IDirect3DViewport2** interface, this method has been superseded by the **IDirect3DViewport2::SetViewport2** method.

```
HRESULT SetViewport(
  LPD3DVIEWPORT lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT** structure that contains the new viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

See Also

IDirect3DViewport2::GetViewport, **IDirect3DViewport2::SetViewport2**

IDirect3DViewport2::SetViewport2

The **IDirect3DViewport2::SetViewport2** method sets the viewport registers of the viewport.

```
HRESULT SetViewport2(
  LPD3DVIEWPORT2 lpData
);
```

Parameters

lpData
Address of a **D3DVIEWPORT2** structure that contains the new viewport.

Return Values

If the method succeeds, the return value is D3D_OK.

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

See Also

IDirect3DViewport2::GetViewport2

IDirect3DViewport2::TransformVertices

The **IDirect3DViewport2::TransformVertices** method transforms a set of vertices by the transformation matrix.

```
HRESULT TransformVertices(
  DWORD dwVertexCount,
  LPD3DTRANSFORMDATA lpData,
  DWORD dwFlags,
  LPDWORD lpOffscreen
);
```

Parameters

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.

Return Values

If the method succeeds, the return value is D3D_OK .

If the method fails, the return value may be one of the following values:

DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS

Remarks

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 **D3DLVERTEX** 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 2-D 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.

This method is unchanged from its implementation in the **IDirect3DViewport** interface.

D3D_OVERLOADS

C++ programmers who define D3D_OVERLOADS can use the extensions documented here to simplify their code in Direct3D Immediate Mode applications. D3D_OVERLOADS was introduced with DirectX® 5. This section is a reference to the D3D_OVERLOADS extensions.

These extensions must be defined with C++ linkage. If D3D_OVERLOADS is defined and the inclusion of D3dtypes.h or D3d.h is surrounded by extern "C", link errors will result. For example, the following syntax would generate link errors because of C linkage of D3D_OVERLOADS functionality:

```
#define D3D_OVERLOADS
extern "C" {
#include <d3d.h>
};
```

The D3D_OVERLOADS extensions can be organized into the following groups:

Constructors

D3DLVERTEX
D3DTLVERTEX
D3DVECTOR
D3DVERTEX

Operators

Access Grant Operators
Addition Operator
Assignment Operators
Bitwise Equality Operator
D3DMATRIX
Division Operator
Multiplication Operator
Subtraction Operator
Unary Operators
Vector Dominance Operators

Helper functions

CrossProduct
DotProduct
Magnitude
Max
Maximize
Min
Minimize
Normalize
SquareMagnitude

D3D_OVERLOADS Constructors

This section contains reference information for the constructors provided by the D3D_OVERLOADS C++ extensions.

- **D3DLVERTEX**
- **D3DTLVERTEX**
- **D3DVECTOR**
- **D3DVERTEX**

D3DLVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DLVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DLVERTEX() { }
_D3DLVERTEX(const D3DVECTOR& v,
            D3DCOLOR _color, D3DCOLOR _specular,
            float _tu, float _tv)
    { x = v.x; y = v.y; z = v.z; dwReserved = 0;
      color = _color; specular = _specular;
      tu = _tu; tv = _tv;
    }
```

D3DTLVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DTLVERTEX** structure offer a convenient way for C++ programmers to create transformed and lit vertices.

```
_D3DTLVERTEX() { }
_D3DTLVERTEX(const D3DVECTOR& v, float _rhw,
             D3DCOLOR _color, D3DCOLOR _specular,
             float _tu, float _tv)
    { sx = v.x; sy = v.y; sz = v.z; rhw = _rhw;
      color = _color; specular = _specular;
      tu = _tu; tv = _tv;
    }
```

D3DVECTOR Constructors

The D3D_OVERLOADS constructors for the **D3DVECTOR** structure offer a convenient way for C++ programmers to create vectors.

```
_D3DVECTOR() { }
_D3DVECTOR(D3DVALUE f);
_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z);
_D3DVECTOR(const D3DVALUE f[3]);
```

These constructors are defined as follows:

```
inline _D3DVECTOR::_D3DVECTOR(D3DVALUE f)
    {    x = y = z = f; }

inline _D3DVECTOR::_D3DVECTOR(D3DVALUE _x, D3DVALUE _y, D3DVALUE _z)
    {     x = _x; y = _y; z = _z; }

inline _D3DVECTOR::_D3DVECTOR(const D3DVALUE f[3])
    {     x = f[0]; y = f[1]; z = f[2]; }
```

D3DVERTEX Constructors

The D3D_OVERLOADS constructors for the **D3DVERTEX** structure offer a convenient way for C++ programmers to create lit vertices.

```
_D3DVERTEX() { }
_D3DVERTEX(const D3DVECTOR& v, const D3DVECTOR& n, float _tu, float _tv)
    { x = v.x; y = v.y; z = v.z;
      nx = n.x; ny = n.y; nz = n.z;
      tu = _tu; tv = _tv;
    }
```

D3D_OVERLOADS Operators

This section contains reference information for the operators provided by the D3D_OVERLOADS C++ extensions.

Access Grant Operators (D3D_OVERLOADS)

The bracket ("[]") operators are overloaded operators for the D3D_OVERLOADS extensions. You can use empty brackets ("[]") for access grants, "v[0]" to access the x component of a vector, "v[1]" to access the y component, and "v[2]" to access the z component. These operators are defined as follows:

```
const D3DVALUE&operator[](int i) const;
D3DVALUE&operator[](int i);

inline const D3DVALUE&
_D3DVECTOR::operator[](int i) const
{
    return (&x)[i];
}

inline D3DVALUE&
_D3DVECTOR::operator[](int i)
{
    return (&x)[i];
}
```

Addition Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The addition operator is defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator + (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR(v1.x+v2.x, v1.y+v2.y, v1.z+v2.z);
}
```

Assignment Operators (D3D_OVERLOADS)

The assignment operators are overloaded operators for the D3D_OVERLOADS extensions. Both scalar and vector forms of the "*=" and "/=" operators have been implemented. (In the vector form, multiplication and division are memberwise.)

```
_D3DVECTOR& operator += (const _D3DVECTOR& v);
_D3DVECTOR& operator -= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (const _D3DVECTOR& v);
_D3DVECTOR& operator /= (const _D3DVECTOR& v);
_D3DVECTOR& operator *= (D3DVALUE s);
_D3DVECTOR& operator /= (D3DVALUE s);
```

The assignment operators are defined as follows:

```
inline _D3DVECTOR&
_D3DVECTOR::operator += (const _D3DVECTOR& v)
{
   x += v.x;   y += v.y;   z += v.z;
   return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator -= (const _D3DVECTOR& v)
{
   x -= v.x;   y -= v.y;   z -= v.z;
   return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator *= (const _D3DVECTOR& v)
{
   x *= v.x;   y *= v.y;   z *= v.z;
   return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (const _D3DVECTOR& v)
{
   x /= v.x;   y /= v.y;   z /= v.z;
   return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator *= (D3DVALUE s)
{
   x *= s;   y *= s;   z *= s;
   return *this;
}

inline _D3DVECTOR&
_D3DVECTOR::operator /= (D3DVALUE s)
{
   x /= s;   y /= s;   z /= s;
   return *this;
```

```
}
```

Bitwise Equality Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The bitwise-equality operator is defined as follows:

```
int operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline int
operator == (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return v1.x==v2.x && v1.y==v2.y && v1.z == v2.z;
}
```

D3DMATRIX (D3D_OVERLOADS)

The D3D_OVERLOADS implementation of the **D3DMATRIX** structure implements a parentheses ("()") operator. This operator offers convenient access to values in the matrix for C++ programmers. Instead of having to refer to the structure members by name, C++ programmers can refer to them by row and column number, and simply index these numbers as needed.

```
typedef struct _D3DMATRIX {
#if (defined __cplusplus) && (defined D3D_OVERLOADS)
    union {
        struct {
#endif

            D3DVALUE        _11, _12, _13, _14;
            D3DVALUE        _21, _22, _23, _24;
            D3DVALUE        _31, _32, _33, _34;
            D3DVALUE        _41, _42, _43, _44;

#if (defined __cplusplus) && (defined D3D_OVERLOADS)
        };
        D3DVALUE m[4][4];
    };
    _D3DMATRIX() { }

    D3DVALUE& operator()(int iRow, int iColumn) { return m[iRow][iColumn]; }
    const D3DVALUE& operator()(int iRow, int iColumn) const { return m[iRow]
[iColumn]; }
#endif
} D3DMATRIX, *LPD3DMATRIX;
```

See Also

D3DMATRIX

Division Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator have been implemented. The division operator is defined as follows:

```
_D3DVECTOR operator / (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator / (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR(v1.x/v2.x, v1.y/v2.y, v1.z/v2.z);
}

inline _D3DVECTOR
operator / (const _D3DVECTOR& v, D3DVALUE s)
{
   return _D3DVECTOR(v.x/s, v.y/s, v.z/s);
}
```

Multiplication Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. Both scalar and vector forms of this operator have been implemented. The multiplication operator is defined as follows:

```
_D3DVECTOR operator * (const _D3DVECTOR& v, D3DVALUE s);
_D3DVECTOR operator * (D3DVALUE s, const _D3DVECTOR& v);
_D3DVECTOR operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator * (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR(v1.x*v2.x, v1.y*v2.y, v1.z*v2.z);
}

inline _D3DVECTOR
operator * (const _D3DVECTOR& v, D3DVALUE s)
{
   return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}

inline _D3DVECTOR
operator * (D3DVALUE s, const _D3DVECTOR& v)
{
   return _D3DVECTOR(s*v.x, s*v.y, s*v.z);
}
```

Subtraction Operator (D3D_OVERLOADS)

This binary operator is an overloaded operator for the D3D_OVERLOADS extensions. The subtraction operator is defined as follows:

```
_D3DVECTOR operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2);

inline _D3DVECTOR
operator - (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z);
}
```

Unary Operators (D3D_OVERLOADS)

The unary operators are overloaded operators for the D3D_OVERLOADS extensions. The unary operators are defined as follows:

```
_D3DVECTOR operator + (const _D3DVECTOR& v);
_D3DVECTOR operator - (const _D3DVECTOR& v);

inline _D3DVECTOR
operator + (const _D3DVECTOR& v)
{
   return v;
}

inline _D3DVECTOR
operator - (const _D3DVECTOR& v)
{
   return _D3DVECTOR(-v.x, -v.y, -v.z);
}
```

Vector Dominance Operators (D3D_OVERLOADS)

These binary operators are overloaded operators for the D3D_OVERLOADS extensions. Vector v1 dominates vector v2 if any component of v1 is greater than the corresponding component of v2. Therefore, it is possible for neither of the two specified vectors to dominate the other.

```
int operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
int operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

The vector-dominance operators are defined as follows:

```
inline int
operator < (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return v1[0] < v2[0] && v1[1] < v2[1] && v1[2] < v2[2];
}

inline int
operator <= (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return v1[0] <= v2[0] && v1[1] <= v2[1] && v1[2] <= v2[2];
}
```

D3D_OVERLOADS Helper Functions

This section contains reference information for the helper functions provided by the D3D_OVERLOADS C++ extensions.

- **CrossProduct**
- **DotProduct**
- **Magnitude**
- **Max**
- **Maximize**
- **Min**
- **Minimize**
- **Normalize**
- **SquareMagnitude**

CrossProduct

This helper function returns the cross product of the specified vectors. **CrossProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
CrossProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
    _D3DVECTOR result;

    result[0] = v1[1] * v2[2] - v1[2] * v2[1];
    result[1] = v1[2] * v2[0] - v1[0] * v2[2];
    result[2] = v1[0] * v2[1] - v1[1] * v2[0];

    return result;
}
```

See Also

DotProduct

DotProduct

This helper function returns the dot product of the specified vectors. **DotProduct** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline D3DVALUE
DotProduct (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return v1.x*v2.x + v1.y * v2.y + v1.z*v2.z;
}
```

See Also

CrossProduct

Magnitude

This helper function returns the absolute value of the specified vector. **Magnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Magnitude (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Magnitude (const _D3DVECTOR& v)
{
   return (D3DVALUE) sqrt(SquareMagnitude(v));
}
```

See Also

SquareMagnitude

Max

This helper function returns the maximum component of the specified vector. **Max** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Max (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Max (const _D3DVECTOR& v)
{
   D3DVALUE ret = v.x;
   if (ret < v.y) ret = v.y;
   if (ret < v.z) ret = v.z;
   return ret;
}
```

See Also

Min

Maximize

This helper function returns a vector that is made up of the largest components of the two specified vectors. **Maximize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
Maximize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR( v1[0] > v2[0] ? v1[0] : v2[0],
                      v1[1] > v2[1] ? v1[1] : v2[1],
                      v1[2] > v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You could use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
    void
    ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min,
D3DVECTOR *max)
    {
        int i;
        *min = *max = pts[0];
        for (i = 1; i < N; i += 1)
        {
            *min = Minimize(*min, pts[i]);
            *max = Maximize(*max, pts[i]);
        }
    }
```

See Also

Minimize

Min

This helper function returns the minimum component of the specified vector. **Min** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE Min (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
Min (const _D3DVECTOR& v)
{
   D3DVALUE ret = v.x;
   if (v.y < ret) ret = v.y;
   if (v.z < ret) ret = v.z;
   return ret;
}
```

See Also

Max

Minimize

This helper function returns a vector that is made up of the smallest components of the two specified vectors. **Minimize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2);
```

This function is defined as follows:

```
inline _D3DVECTOR
Minimize (const _D3DVECTOR& v1, const _D3DVECTOR& v2)
{
   return _D3DVECTOR( v1[0] < v2[0] ? v1[0] : v2[0],
                      v1[1] < v2[1] ? v1[1] : v2[1],
                      v1[2] < v2[2] ? v1[2] : v2[2]);
}
```

Remarks

You could use the **Maximize** and **Minimize** functions to compute the bounding box for a set of points, in a function that looks like this:

```
    void
    ComputeBoundingBox(const D3DVECTOR *pts, int N, D3DVECTOR *min,
D3DVECTOR *max)
    {
        int i;
        *min = *max = pts[0];
        for (i = 1; i < N; i += 1)
        {
            *min = Minimize(*min, pts[i]);
            *max = Maximize(*max, pts[i]);
        }
    }
```

See Also

Maximize

Normalize

This helper function returns the normalized version of the specified vector (that is, a unit-length vector with the same direction as the source). **Normalize** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
_D3DVECTOR Normalize (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline _D3DVECTOR
Normalize (const _D3DVECTOR& v)
{
   return v / Magnitude(v);
}
```

SquareMagnitude

This helper function returns the square of the absolute value of the specified vector. **SquareMagnitude** is part of the suite of extra functionality that is available to C++ programmers who define D3D_OVERLOADS.

```
D3DVALUE SquareMagnitude (const _D3DVECTOR& v);
```

This function is defined as follows:

```
inline D3DVALUE
SquareMagnitude (const _D3DVECTOR& v)
{
   return v.x*v.x + v.y*v.y + v.z*v.z;
}
```

See Also

Magnitude

Macros

This section contains reference information for the macros provided by Direct3D's Immediate Mode.

- **D3DDivide**
- **D3DMultiply**
- **D3DRGB**
- **D3DRGBA**
- **D3DSTATE_OVERRIDE**
- **D3DVAL**
- **D3DVALP**
- **RGB_GETBLUE**
- **RGB_GETGREEN**
- **RGB_GETRED**
- **RGB_MAKE**
- **RGB_TORGBA**
- **RGBA_GETALPHA**
- **RGBA_GETBLUE**
- **RGBA_GETGREEN**
- **RGBA_GETRED**
- **RGBA_MAKE**
- **RGBA_SETALPHA**
- **RGBA_TORGB**

D3DDivide

The **D3DDivide** macro divides two values.

```
D3DDivide(a, b)    (float)((double) (a) / (double) (b))
```

Parameters

a and *b*
Dividend and divisor in the expression, respectively.

Return Values

The macros returns the quotient of the division.

See Also

D3DMultiply

D3DMultiply

The **D3DMultiply** macro multiplies two values.

```
D3DMultiply(a, b)     ((a) * (b))
```

Parameters

a and *b*
Values to be multiplied.

Return Values

The macros returns the product of the multiplication.

See Also

D3DDivide

D3DRGB

The **D3DRGB** macro initializes a color with the supplied RGB values.

```
D3DRGB(r, g, b) \
    (0xff000000L | ( ((long)((r) * 255)) << 16) | \
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, and *b*

Red, green, and blue components of the color. These should be floating-point values in the range 0 through 1.

Return Values

The macros returns the **D3DCOLOR** value corresponding to the supplied RGB values.

See Also

D3DRGBA

D3DRGBA

The **D3DRGBA** macro initializes a color with the supplied RGBA values.

```
D3DRGBA(r, g, b, a) \
    ((((long)((a) * 255)) << 24) | (((long)((r) * 255)) << 16) |
    (((long)((g) * 255)) << 8) | (long)((b) * 255))
```

Parameters

r, *g*, *b*, and *a*
Red, green, blue, and alpha components of the color.

Return Values

The macros returns the **D3DCOLOR** value corresponding to the supplied RGBA values.

See Also

D3DRGB

D3DSTATE_OVERRIDE

The **D3DSTATE_OVERRIDE** macro overrides the state of the rasterization, lighting, or transformation module. Applications can use this macro to lock and unlock a state.

```
D3DSTATE_OVERRIDE(type) ((DWORD) (type) + D3DSTATE_OVERRIDE_BIAS)
```

Parameters

type

State to override. This parameter should be one of the members of the **D3DTRANSFORMSTATETYPE**, **D3DLIGHTSTATETYPE**, or **D3DRENDERSTATETYPE** enumerated types.

Return Values

No return value.

Remarks

An application might, for example, use the **STATE_DATA** macro (defined in the D3dmacs.h header file in the Misc directory of the DirectX 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

The **D3DVAL** macro creates a value whose type is **D3DVALUE**.

```
D3DVAL(val)     ((float)val)
```

Parameters

val
Value to be converted.

Return Values

The macros returns the converted value.

See Also

D3DVALP

D3DVALP

The **D3DVALP** macro creates a value of the specified precision.

```
D3DVALP(val, prec)    ((float)val)
```

Parameters

val
Value to be converted.

prec
Ignored.

Return Values

The macros returns the converted value.

Remarks

The precision, as implemented by the **D3DVAL** macro, is 16 bits for the fractional part of the value.

See Also

D3DVAL

RGB_GETBLUE

The **RGB_GETBLUE** macro retrieves the blue component of a **D3DCOLOR** value.

```
RGB_GETBLUE(rgb)     ((rgb) & 0xff)
```

Parameters

rgb

Color index from which the blue component is retrieved.

Return Values

Returns the blue component.

RGB_GETGREEN

The **RGB_GETGREEN** macro retrieves the green component of a **D3DCOLOR** value.

```
RGB_GETGREEN(rgb)     (((rgb) >> 8) & 0xff)
```

Parameters

rgb

Color index from which the green component is retrieved.

Return Values

The macros returns the green component.

RGB_GETRED

The **RGB_GETRED** macro retrieves the red component of a **D3DCOLOR** value.

```
RGB_GETRED(rgb)     (((rgb) >> 16) & 0xff)
```

Parameters

rgb

Color index from which the red component is retrieved.

Return Values

The macros returns the red component.

RGB_MAKE

The **RGB_MAKE** macro creates an RGB color from supplied values.

```
RGB_MAKE(r, g, b)     ((D3DCOLOR) (((r) << 16) | ((g) << 8) | (b)))
```

Parameters

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.

Return Values

The macros returns the color.

RGB_TORGBA

The **RGB_TORGBA** macro creates an RGBA color from a supplied RGB color.

```
RGB_TORGBA(rgb)     ((D3DCOLOR) ((rgb) | 0xff000000))
```

Parameters

rgb
RGB color to be converted to an RGBA color.

Return Values

Returns the RGBA color.

See Also

RGBA_TORGB

RGBA_GETALPHA

The **RGB_GETALPHA** macro retrieves the alpha component of an RGBA **D3DCOLOR** value.

```
RGBA_GETALPHA(rgb)     ((rgb) >> 24)
```

Parameters

rgb

Color index from which the alpha component is retrieved.

Return Values

The macros returns the alpha component.

RGBA_GETBLUE

The **RGBA_GETBLUE** macro retrieves the blue component of an RGBA **D3DCOLOR** value.

```
RGB_GETBLUE(rgb)     ((rgb) & 0xff)
```

Parameters

rgb

Color index from which the blue component is retrieved.

Return Values

The macros returns the blue component.

RGBA_GETGREEN

The **RGBA_GETGREEN** macro retrieves the green component of an RGBA **D3DCOLOR** value.

```
RGB_GETGREEN(rgb)     (((rgb) >> 8) & 0xff)
```

Parameters

rgb

Color index from which the green component is retrieved.

Return Values

The macros returns the green component.

RGBA_GETRED

The **RGBA_GETRED** macro retrieves the red component of an RGBA **D3DCOLOR** value.

```
RGB_GETRED(rgb)     (((rgb) >> 16) & 0xff)
```

Parameters

rgb

Color index from which the red component is retrieved.

Return Values

The macros returns the red component.

RGBA_MAKE

The **RGBA_MAKE** macro creates an RGBA **D3DCOLOR** value from supplied red, green, blue, and alpha components.

```
RGBA_MAKE(r, g, b, a)    \
    ((D3DCOLOR) (((a) << 24) | ((r) << 16) | ((g) << 8) | (b)))
```

Parameters

r, *g*, *b*, and *a*

Red, green, blue, and alpha components of the RGBA color to be created.

Return Values

The macros returns the color.

RGBA_SETALPHA

The **RGBA_SETALPHA** macro sets the alpha component of an RGBA **D3DCOLOR** value.

```
RGBA_SETALPHA(rgba, x)    (((x) << 24) | ((rgba) & 0x00ffffff))
```

Parameters

rgba

RGBA color for which the alpha component will be set.

x

Value of alpha component to be set.

Return Values

The macros returns the RGBA color whose alpha component has been set.

RGBA_TORGB

The **RGBA_TORGB** macro creates an RGB **D3DCOLOR** value from a supplied RGBA **D3DCOLOR** value by stripping off the alpha component of the color.

```
RGBA_TORGB(rgba)     ((D3DCOLOR) ((rgba) & 0xffffff))
```

Parameters

rgba

RGBA color to be converted to an RGB color.

Return Values

The macros returns the RGB color.

See Also

RGB_TORGBA

Callback Functions

This section contains reference information for the callback functions you may need to implement when you work with Direct3D Immediate Mode.

- **D3DENUMDEVICESCALLBACK**
- **D3DENUMTEXTUREFORMATSCALLBACK**
- **D3DVALIDATECALLBACK**

D3DENUMDEVICESCALLBACK

D3DENUMDEVICESCALLBACK is the prototype definition for the callback function to enumerate installed Direct3D devices.

```
typedef HRESULT (FAR PASCAL * LPD3DENUMDEVICESCALLBACK)
  (LPGUID lpGuid,
  LPSTR lpDeviceDescription,
  LPSTR lpDeviceName,
  LPD3DDEVICEDESC lpD3DHWDeviceDesc,
  LPD3DDEVICEDESC lpD3DHELDeviceDesc,
  LPVOID lpUserArg
);
```

Parameters

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 of a **D3DDEVICEDESC** structure that contains the hardware capabilities of the Direct3D device.

lpD3DHELDeviceDesc
Address of a **D3DDEVICEDESC** structure that contains the emulated capabilities of the Direct3D device.

lpUserArg
Address of application-defined data passed to this callback function.

Return Values

Applications should return one of the following values:

D3DENUMRET_CANCEL
Cancel the enumeration.

D3DENUMRET_OK
Continue the enumeration.

Remarks

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

D3DENUMTEXTUREFORMATSCALLBACK is the prototype definition for the callback function to enumerate texture formats.

```
typedef HRESULT (WINAPI* LPD3DENUMTEXTUREFORMATSCALLBACK)
  (LPDDSURFACEDESC lpDdsd,
  LPVOID lpUserArg
);
```

Parameters

lpDdsd

Address of a **DDSURFACEDESC** structure containing the texture information.

lpUserArg

Address of application-defined data passed to this callback function.

Return Values

Applications should return one of the following values:

D3DENUMRET_CANCEL

Cancel the enumeration.

D3DENUMRET_OK

Continue the enumeration.

Remarks

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

D3DVALIDATECALLBACK is the application-defined callback function supplied when an application calls the **IDirect3DExecuteBuffer::Validate** method. The **IDirect3DExecuteBuffer::Validate** method is not currently implemented.

IDirect3DExecuteBuffer::Validate is a debugging routine that checks the execute buffer and returns an offset into the buffer when any errors are encountered.

```
typedef HRESULT (WINAPI* LPD3DVALIDATECALLBACK)
  (LPVOID lpUserArg,
  DWORD dwOffset
);
```

Parameters

lpUserArg

Address of application-defined data passed to this callback function.

dwOffset

Offset into the execute buffer at which the system found an error.

Remarks

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.

Structures

This section contains information about the following structures used with Direct3D Immediate Mode.

- **D3DBRANCH**
- **D3DCLIPSTATUS**
- **D3DCOLORVALUE**
- **D3DDEVICEDESC**
- **D3DEXECUTEBUFFERDESC**
- **D3DEXECUTEDATA**
- **D3DFINDDEVICERESULT**
- **D3DFINDDEVICESEARCH**
- **D3DHVERTEX**
- **D3DINSTRUCTION**
- **D3DLIGHT2**
- **D3DLIGHTDATA**
- **D3DLIGHTINGCAPS**
- **D3DLIGHTINGELEMENT**
- **D3DLINE**
- **D3DLINEPATTERN**
- **D3DLVERTEX**
- **D3DMATERIAL**
- **D3DMATRIX**
- **D3DMATRIXLOAD**
- **D3DMATRIXMULTIPLY**
- **D3DPICKRECORD**
- **D3DPOINT**
- **D3DPRIMCAPS**
- **D3DPROCESSVERTICES**
- **D3DRECT**
- **D3DSPAN**
- **D3DSTATE**
- **D3DSTATS**
- **D3DSTATUS**
- **D3DTEXTURELOAD**
- **D3DTLVERTEX**
- **D3DTRANSFORMCAPS**
- **D3DTRANSFORMDATA**
- **D3DTRIANGLE**
- **D3DVECTOR**
- **D3DVERTEX**
- **D3DVIEWPORT**
- **D3DVIEWPORT2**

D3DBRANCH

The **D3DBRANCH** structure performs conditional operations inside an execute buffer. This structure is a forward-branch structure.

```
typedef struct _D3DBRANCH {
    DWORD dwMask;
    DWORD dwValue;
    BOOL  bNegate;
    DWORD dwOffset;
} D3DBRANCH, *LPD3DBRANCH;
```

Members

dwMask

Bitmask for the branch. This mask is combined with the driver-status mask by using the bitwise **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.

D3DCLIPSTATUS

The **D3DCLIPSTATUS** structure describes the current clip status and extents of the clipping region. This structure was introduced in DirectX 5.

```
typedef struct _D3DCLIPSTATUS {
    DWORD dwFlags;
    DWORD dwStatus;
    float minx, maxx;
    float miny, maxy;
    float minz, maxz;
} D3DCLIPSTATUS, *LPD3DCLIPSTATUS;
```

Members

dwFlags

Flags describing whether this structure describes 2-D extents, 3-D extents, or the clip status. This member can be a combination of the following flags:

D3DCLIPSTATUS_STATUS

The structure describes the current clip status.

D3DCLIPSTATUS_EXTENTS2

The structure describes the current 2-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS3.

D3DCLIPSTATUS_EXTENTS3

The structure describes the current 3-D extents. This flag cannot be combined with D3DCLIPSTATUS_EXTENTS2.

dwStatus

Describes the current clip status. For a list of the available driver-status masks, see the **dwStatus** member of the **D3DSTATUS** structure.

minx, maxx, miny, maxy, minz, maxz

x, y, and z extents of the current clipping region.

See Also

IDirect3DDevice2::GetClipStatus, **IDirect3DDevice2::SetClipStatus**

D3DCOLORVALUE

The **D3DCOLORVALUE** structure describes color values for the **D3DLIGHT2** and **D3DMATERIAL** structures.

```
typedef struct _D3DCOLORVALUE {
    union {
        D3DVALUE r;
        D3DVALUE dvR;
    };
    union {
        D3DVALUE g;
        D3DVALUE dvG;
    };
    union {
        D3DVALUE b;
        D3DVALUE dvB;
    };
    union {
        D3DVALUE a;
        D3DVALUE dvA;
    };
} D3DCOLORVALUE;
```

Members

dvR, **dvG**, **dvB**, and **dvA**

Values of the **D3DVALUE** type specifying the red, green, blue, and alpha components of a color. These values generally range from 0 to 1, with 0 being black.

Remarks

You can set the members of this structure to values outside the range of 0 to 1 to implement some unusual effects. Values greater than 1 produce strong lights that tend to wash out a scene. Negative values produce dark lights, which actually remove light from a scene. For more information, see Colored Lights.

D3DDEVICEDESC

The **D3DDEVICEDESC** structure contains a description of the current device. This structure is used to query the current device by such methods as **IDirect3DDevice2::GetCaps**.

```
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;
    DWORD            dwMinTextureWidth, dwMinTextureHeight;
    DWORD            dwMaxTextureWidth, dwMaxTextureHeight;
    DWORD            dwMinStippleWidth, dwMaxStippleWidth;
    DWORD            dwMinStippleHeight, dwMaxStippleHeight;
} D3DDEVICEDESC, *LPD3DDEVICEDESC;
```

Members

dwSize

Size, in bytes, of this structure. You can use the **D3DDEVICEDESCSIZE** constant for this value. This member must be initialized before the structure is used.

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_DEVICERENDERBITDEPTH

The **dwDeviceRenderBitDepth** member is valid.

D3DDD_DEVICEZBUFFERBITDEPTH

The **dwDeviceZBufferBitDepth** 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_CANRENDERAFTERFLIP

Device can queue rendering commands after a page flip. Applications should not change their behavior if this flag is set; this capability simply means that the device is relatively fast.

This flag was introduced in DirectX 5.

D3DDEVCAPS_DRAWPRIMTLVERTEX

Device exports a DrawPrimitive-aware HAL.

This flag was introduced in DirectX 5.

D3DDEVCAPS_EXECUTESYSTEMMEMORY

Device can use execute buffers from system memory.

D3DDEVCAPS_EXECUTEVIDEOMEMORY

Device can use execute buffer 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_TEXTURENONLOCALVIDMEM

Device can retrieve textures from nonlocal video (AGP) memory.

This flag was introduced in DirectX 5. For more information about AGP memory, see Using Non-local Video Memory Surfaces in the DirectDraw documentation.

D3DDEVCAPS_TEXTURESYSTEMMEMORY

Device can retrieve textures from system memory.

D3DDEVCAPS_TEXTUREVIDEOMEMORY

Device can retrieve 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 3-D 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 or more 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.

dwMinTextureWidth, dwMinTextureHeight
Minimum texture width and height for this device. These members were introduced in DirectX 5.

dwMaxTextureWidth, dwMaxTextureHeight
Maximum texture width and height for this device. These members were introduced in DirectX 5.

dwMinStippleWidth, dwMaxStippleWidth
Minimum and maximum width of the stipple pattern for this device. These members were introduced in DirectX 5.

dwMinStippleHeight, dwMaxStippleHeight
Minimum and maximum height of the stipple pattern for this device. These members were introduced in DirectX 5.

See Also

D3DCOLORMODEL, **D3DFINDDEVICERESULT**, **D3DLIGHTINGCAPS**, **D3DPRIMCAPS**, **D3DTRANSFORMCAPS**

D3DEXECUTEBUFFERDESC

The **D3DEXECUTEBUFFERDESC** structure describes the execute buffer for such methods as **IDirect3DDevice::CreateExecuteBuffer** and **IDirect3DExecuteBuffer::Lock**.

```
typedef struct _D3DExecuteBufferDesc {
    DWORD  dwSize;
    DWORD  dwFlags;
    DWORD  dwCaps;
    DWORD  dwBufferSize;
    LPVOID lpData;
} D3DEXECUTEBUFFERDESC;
typedef D3DEXECUTEBUFFERDESC *LPD3DEXECUTEBUFFERDESC;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized before the structure is used.

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_MEM

A logical **OR** of D3DDEBCAPS_SYSTEMMEMORY and D3DDEBCAPS_VIDEOMEMORY.

D3DDEBCAPS_SYSTEMMEMORY

The execute buffer data resides in system memory.

D3DDEBCAPS_VIDEOMEMORY

The execute buffer data resides in device memory.

dwBufferSize

Size of the execute buffer, in bytes.

lpData

Address of the buffer data.

D3DEXECUTEDATA

The **D3DEXECUTEDATA** structure 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.

```
typedef struct _D3DEXECUTEDATA {
    DWORD     dwSize;
    DWORD     dwVertexOffset;
    DWORD     dwVertexCount;
    DWORD     dwInstructionOffset;
    DWORD     dwInstructionLength;
    DWORD     dwHVertexOffset;
    D3DSTATUS dsStatus;
} D3DEXECUTEDATA, *LPD3DEXECUTEDATA;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

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.

See Also

D3DSTATUS

D3DFINDDEVICERESULT

The **D3DFINDDEVICERESULT** structure identifies a device an application has found by calling the **IDirect3D2::FindDevice** method.

```
typedef struct _D3DFINDDEVICERESULT {
    DWORD         dwSize;
    GUID          guid;
    D3DDEVICEDESC ddHwDesc;
    D3DDEVICEDESC ddSwDesc;
} D3DFINDDEVICERESULT, *LPD3DFINDDEVICERESULT;
```

Members

dwSize
Size, in bytes, of the structure. This member must be initialized before the structure is used.

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.

See Also

D3DFINDDEVICESEARCH

D3DFINDDEVICESEARCH

The **D3DFINDDEVICESEARCH** structure specifies the characteristics of a device an application wants to find. This structure is used in calls to the **IDirect3D2::FindDevice** method.

```
typedef struct _D3DFINDDEVICESEARCH {
    DWORD         dwSize;
    DWORD         dwFlags;
    BOOL          bHardware;
    D3DCOLORMODEL dcmColorModel;
    GUID          guid;
    DWORD         dwCaps;
    D3DPRIMCAPS   dpcPrimCaps;
} D3DFINDDEVICESEARCH, *LPD3DFINDDEVICESEARCH;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

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

Reserved.

dpcPrimCaps

Specifies a **D3DPRIMCAPS** structure defining the device's capabilities for each primitive type.

See Also

D3DFINDDEVICERESULT

D3DHVERTEX

The **D3DHVERTEX** structure defines a homogeneous vertex used when the application is supplying screen coordinate data that needs clipping. This structure is part of the **D3DTRANSFORMDATA** structure.

```
typedef struct _D3DHVERTEX {
    DWORD        dwFlags;
    union {
        D3DVALUE hx;
        D3DVALUE dvHX;
    };
    union {
        D3DVALUE hy;
        D3DVALUE dvHY;
    };
    union {
        D3DVALUE hz;
        D3DVALUE dvHZ;
    };
} D3DHVERTEX, *LPD3DHVERTEX;
```

Members

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

The **D3DINSTRUCTION** structure 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.

```
typedef struct _D3DINSTRUCTION {
    BYTE bOpcode;
    BYTE bSize;
    WORD wCount;
} D3DINSTRUCTION, *LPD3DINSTRUCTION;
```

Members

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.

D3DLIGHT2

The **D3DLIGHT2** structure defines the light type in calls to methods such as **IDirect3DLight::SetLight** and **IDirect3DLight::GetLight**.

For DirectX 5, this structure supersedes the **D3DLIGHT** structure. **D3DLIGHT2** is identical to **D3DLIGHT** except for the addition of the **dwFlags** member. In addition, the **dvAttenuation** members are interpreted differently in **D3DLIGHT2** than they were for **D3DLIGHT**.

```
typedef struct _D3DLIGHT2 {
    DWORD         dwSize;
    D3DLIGHTTYPE  dltType;
    D3DCOLORVALUE dcvColor;
    D3DVECTOR     dvPosition;
    D3DVECTOR     dvDirection;
    D3DVALUE      dvRange;
    D3DVALUE      dvFalloff;
    D3DVALUE      dvAttenuation0;
    D3DVALUE      dvAttenuation1;
    D3DVALUE      dvAttenuation2;
    D3DVALUE      dvTheta;
    D3DVALUE      dvPhi;
    DWORD         dwFlags;        // new member for DirectX 5
} D3DLIGHT2, *LPD3DLIGHT2;
```

Members

dwSize
Size, in bytes, of this structure. You must specify a value for this member. Direct3D uses the specified size to determine whether this is a **D3DLIGHT** or a **D3DLIGHT2** 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. In ramp mode, the color is converted to a gray scale.

dvPosition
Position of the light in world space. This member has no meaning for directional lights and is ignored in that case.

dvDirection
Direction the light is pointing in world space. This member only has meaning for directional and spotlights. This vector need not be normalized but it should have a non-zero length.

dvRange
Distance beyond which the light has no effect. The maximum allowable value for this member is D3DLIGHT_RANGE_MAX, which is defined as the square root of FLT_MAX. This member does not affect directional lights.

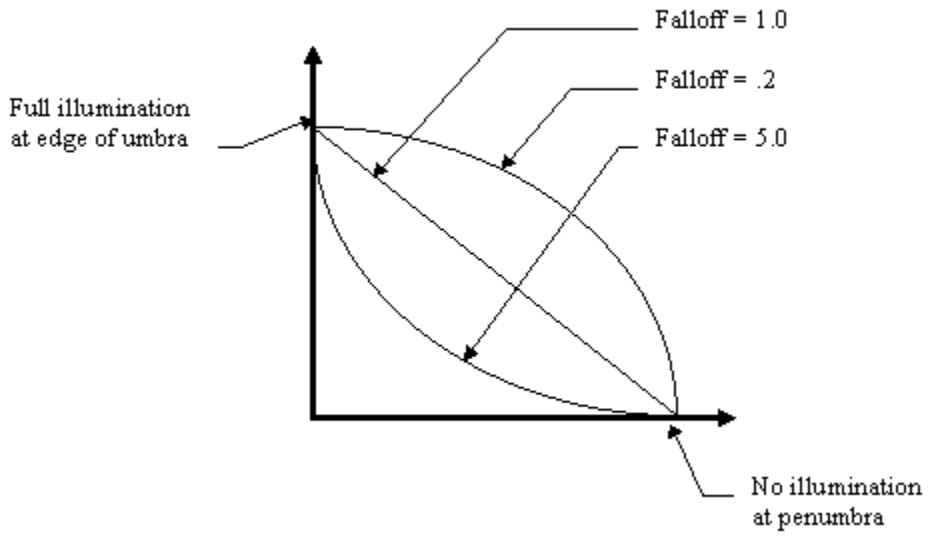
dvFalloff
Decrease in illumination between a spotlight's umbra (the angle specified by the **dvTheta** member) and the outer edge of the penumbra (the angle specified by the **dvPhi** member). This feature was implemented for DirectX 5.

The intensity of the light at any point in the penumbra is described by the following equation:

$$Light \times \cos^{falloff}\left[\frac{\pi}{2}\left|\frac{2\,rho - dvTheta}{dvPhi - dvTheta}\right|\right]$$

In this equation, *rho* is the angle between the axis of the spotlight and the illuminated point.

A value of 1.0 specifies linear falloff from the umbra to the penumbra. If the value is anything other than 1.0, it is used as an exponent to shape the curve. Values greater than 1.0 cause the light to fall off quickly at first and then fade slowly to the penumbra. Values which are less than 1.0 create the opposite effect. The following graph shows the affect of changing these values:



The effect of falloff on the lighting is subtle. Furthermore, a small performance penalty is incurred by shaping the falloff curve. For these reasons, most developers set this value to 1.0.

dvAttenuation0 through **dvAttenuation2**

Values specifying how a light's intensity changes over distance. (Attenuation does not affect directional lights.) In the **D3DLIGHT2** structure these values are interpreted differently than they were for the **D3DLIGHT** structure.

The distance from the light to the vertex is normalized to the range by the following formula:

```
distance = (range-distance)/range
```

This results in the distance value being from 1.0 at the light to 0.0 at the light's full range. Then the combined intensity factor of the light is calculated using the following formula:

```
intensity = dvAttenuation0 +
    dvAttenuation1 * distance +
    dvAttenuation2 * distance squared
```

This intensity factor is then multiplied by the light color to produce the final intensity of the light.

Setting the attenuation values to 1,0,0 produces a light that doesn't change over distance. Setting the values to 0,1,0 produces a light that is at full intensity at the light, zero intensity at the light's range, and that declines linearly between the two extremes. The values 0,0,1 produce a light that mimics the standard "1/distance squared" falloff rate that should be familiar from introductory physics classes. (The differences in this last case are that the curve is softer and that the intensity of the light doesn't go to infinity at the light source.)

You can use various combinations of values to create unique lights. You can even use negative values – this is another way to achieve a dark light effect. Just as when you use negative values for colors, when you are in ramp mode you cannot use dark lights to produce anything darker than the current setting for the ambient light.

dvTheta

Angle, in radians, of the spotlight's umbra–that is, the fully illuminated spotlight cone. This value must be less than pi radians.

dvPhi

Angle, in radians, defining the outer edge of the spotlight's penumbra. Points outside this cone are not lit by the spotlight. This value must be between 0 and the value specified for the **dvTheta** member.

dwFlags

A combination of the following performance-related flags. This member is new for DirectX 5.

D3DLIGHT_ACTIVE	Enables the light. This flag must be set to enable the light; if it is not set, the light is ignored.
D3DLIGHT_NO_SPECULAR	Turns off specular highlights for the light.

Remarks

In the **D3DLIGHT** structure, the affects of the attenuation settings were difficult to predict; developers were encouraged to experiment with the settings until they achieved the desired result. For **D3DLIGHT2**, it is much easier to work with lighting attenuation.

For more information about lights, see Lights and IDirect3DLight.

See Also

D3DLIGHTTYPE

D3DLIGHTDATA

The **D3DLIGHTDATA** structure describes the points to be lit and resulting colors in calls to the **IDirect3DViewport2::LightElements** method.

```
typedef struct _D3DLIGHTDATA {
    DWORD                dwSize;
    LPD3DLIGHTINGELEMENT lpIn;
    DWORD                dwInSize;
    LPD3DTLVERTEX        lpOut;
    DWORD                dwOutSize;
} D3DLIGHTDATA, *LPD3DLIGHTDATA;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

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

The **D3DLIGHTINGCAPS** structure describes the lighting capabilities of a device. This structure is a member of the **D3DDEVICEDESC** structure.

```
typedef struct _D3DLIGHTINGCAPS {
    DWORD dwSize;
    DWORD dwCaps;
    DWORD dwLightingModel;
    DWORD dwNumLights;
} D3DLIGHTINGCAPS, *LPD3DLIGHTINGCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

dwCaps

Flags describing the capabilities of the lighting module. The following flags are defined:

D3DLIGHTCAPS_DIRECTIONAL

Supports directional lights.

D3DLIGHTCAPS_PARALLELPOINT

Supports parallel point lights.

D3DLIGHTCAPS_POINT

Supports point lights.

D3DLIGHTCAPS_SPOT

Supports spotlights.

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

The **D3DLIGHTINGELEMENT** structure describes the points in model space that will be lit. This structure is part of the **D3DLIGHTDATA** structure.

```
typedef struct _D3DLIGHTINGELEMENT {
    D3DVECTOR dvPosition;
    D3DVECTOR dvNormal;
} D3DLIGHTINGELEMENT, *LPD3DLIGHTINGELEMENT;
```

Members

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.

See Also

D3DLIGHTDATA, **IDirect3DViewport2::LightElements**

D3DLINE

The **D3DLINE** structure describes a line for the **D3DOP_LINE** opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DLINE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
} D3DLINE, *LPD3DLINE;
```

Members

wV1 and **wV2**
Vertex indices.

Remarks

The instruction count defines the number of line segments.

D3DLINEPATTERN

The **D3DLINEPATTERN** structure describes a line pattern. These values are used by the **D3DRENDERSTATE_LINEPATTERN** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef struct _D3DLINEPATTERN {
    WORD wRepeatFactor;
    WORD wLinePattern;
} D3DLINEPATTERN;
```

Members

wRepeatFactor

Number of times to duplicate each series of 1s and 0s specified in the **wLinePattern** member. This repeat factor allows an application to "stretch" the line pattern.

wLinePattern

Bits specifying the line pattern. For example, the following value would produce a dotted line: 1100110011001100.

Remarks

A line pattern specifies how a line is drawn. The line pattern is always the same, no matter where it is started. (This is as opposed to stippling, which affects how objects are rendered; that is, to imitate transparency.)

The line pattern specifies up to a 16-pixel pattern of on and off pixels along the line. The **wRepeatFactor** member specifies how many pixels are repeated for each entry in **wLinePattern**.

D3DLVERTEX

The **D3DLVERTEX** structure defines an untransformed and lit vertex (model coordinates with color). An application should use this structure when the vertex transformations will be handled by Direct3D. This structure contains only data and a color that would be filled by software lighting.

```
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;
```

Members

dvX, **dvY**, and **dvZ**
Values of the **D3DVALUE** type specifying the model 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.

See Also

D3DTLVERTEX, **D3DVERTEX**

D3DMATERIAL

The **D3DMATERIAL** structure specifies material properties in calls to the **IDirect3DMaterial2::GetMaterial** and **IDirect3DMaterial2::SetMaterial** methods.

```
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;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

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.

Remarks

The texture handle specified by the **hTexture** member 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.

To turn off specular highlights for a material, you must set the **dvPower** member to 0—simply setting the specular color components to 0 is not enough.

See Also

IDirect3DMaterial2::GetMaterial, IDirect3DMaterial2::SetMaterial

D3DMATRIX

The **D3DMATRIX** structure describes a matrix for such methods as **IDirect3DDevice::GetMatrix** and **IDirect3DDevice::SetMatrix**.

C++ programmers can use an extended version of this structure that includes a parentheses ("()") operator. For more information, see **D3DMATRIX (D3D_OVERLOADS)**

```
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;
```

Remarks

In Direct3D, the _34 element of a projection matrix cannot be a negative number. If your application needs to use a negative value in this location, it should scale the entire projection matrix by -1, instead.

See Also

IDirect3DDevice::GetMatrix, **IDirect3DDevice::SetMatrix**

D3DMATRIXLOAD

The **D3DMATRIXLOAD** structure describes the operand data for the **D3DOP_MATRIXLOAD** opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DMATRIXLOAD {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix;
} D3DMATRIXLOAD, *LPD3DMATRIXLOAD;
```

Members

hDestMatrix and **hSrcMatrix**

Handles of the destination and source matrices. These values are **D3DMATRIX** structures.

See Also

D3DOPCODE

D3DMATRIXMULTIPLY

The **D3DMATRIXMULTIPLY** structure describes the operand data for the **D3DOP_MATRIXMULTIPLY** opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DMATRIXMULTIPLY {
    D3DMATRIXHANDLE hDestMatrix;
    D3DMATRIXHANDLE hSrcMatrix1;
    D3DMATRIXHANDLE hSrcMatrix2;
} D3DMATRIXMULTIPLY, *LPD3DMATRIXMULTIPLY;
```

Members

hDestMatrix
Handle to 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.

See Also

D3DOPCODE

D3DPICKRECORD

The **D3DPICKRECORD** structure returns information about picked primitives in an execute buffer for the **IDirect3DDevice::GetPickRecords** method.

```
typedef struct _D3DPICKRECORD {
    BYTE     bOpcode;
    BYTE     bPad;
    DWORD    dwOffset;
    D3DVALUE dvZ;
} D3DPICKRECORD, *LPD3DPICKRECORD;
```

Members

bOpcode
Opcode of the picked primitive.

bPad
Pad byte.

dwOffset
Offset from the start of the instruction segment portion of the execute buffer in which the picked primitive was found. (The instruction segment portion of the execute buffer is the part of the execute buffer that follows the vertex list.)

dvZ
Depth of the picked primitive.

Remarks

The x- and y-coordinates of the picked primitive are specified in the call to the **IDirect3DDevice::Pick** method that created the pick records.

See Also

IDirect3DDevice::GetPickRecords, **IDirect3DDevice::Pick**

D3DPOINT

The **D3DPOINT** structure describes operand data for the **D3DOP_POINT** opcode in the in **D3DOPCODE** enumerated type.

```
typedef struct _D3DPOINT {
    WORD wCount;
    WORD wFirst;
} D3DPOINT, *LPD3DPOINT;
```

Members

wCount
Number of points.

wFirst
Index of the first vertex.

Remarks

Points are rendered by using a list of vertices.

See Also

D3DOPCODE

D3DPRIMCAPS

The **D3DPRIMCAPS** structure 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.

```
typedef struct _D3DPrimCaps {
    DWORD dwSize;                // size of structure
    DWORD dwMiscCaps;            // miscellaneous caps
    DWORD dwRasterCaps;          // raster caps
    DWORD dwZCmpCaps;            // z-comparison caps
    DWORD dwSrcBlendCaps;        // source blending caps
    DWORD dwDestBlendCaps;       // destination blending caps
    DWORD dwAlphaCmpCaps;        // alpha-test comparison caps
    DWORD dwShadeCaps;           // shading caps
    DWORD dwTextureCaps;         // texture caps
    DWORD dwTextureFilterCaps;   // texture filtering caps
    DWORD dwTextureBlendCaps;    // texture blending caps
    DWORD dwTextureAddressCaps;  // texture addressing caps
    DWORD dwStippleWidth;        // stipple width
    DWORD dwStippleHeight;       // stipple height
} D3DPRIMCAPS, *LPD3DPRIMCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

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_ANISOTROPY

The device supports anisotropic filtering. For more information, see **D3DRENDERSTATE_ANISOTROPY** in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.

D3DPRASTERCAPS_ANTIALIASEDGES

The device can antialias lines forming the convex outline of objects. For more information, see **D3DRENDERSTATE_EDGEANTIALIAS** in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.

D3DPRASTERCAPS_ANTIALIASSORTDEPENDENT

The device supports antialiasing that is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.

D3DPRASTERCAPS_ANTIALIASSORTINDEPENDENT

The device supports antialiasing that is not dependent on the sort order of the polygons. For more information, see the **D3DANTIALIASMODE** enumerated type.

This flag was introduced in DirectX 5.

D3DPRASTERCAPS_DITHER

The device can dither to improve color resolution.

D3DPRASTERCAPS_FOGRANGE

The device supports range-based fog. In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. For more information, see **D3DRENDERSTATE_RANGEFOGENABLE**.

This flag was introduced in DirectX 5.

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_MIPMAPLODBIAS

The device supports level-of-detail (LOD) bias adjustments. These bias adjustments enable an application to make a mipmap appear crisper or less sharp than it normally would. For more information about LOD bias in mipmaps, see **D3DRENDERSTATE_MIPMAPLODBIAS**.

This flag was introduced in DirectX 5.

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 scan line. 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_ZBIAS

The device supports z-bias values. These are integer values assigned to polygons that allow physically coplanar polygons to appear separate. For more information, see **D3DRENDERSTATE_ZBIAS** in the **D3DRENDERSTATETYPE** structure.

This flag was introduced in DirectX 5.

D3DPRASTERCAPS_ZBUFFERLESSHSR

The device can perform hidden-surface removal without requiring the application to sort polygons, and without requiring the allocation of a z-buffer. This leaves more video memory for textures. The method used to perform hidden-surface removal is

hardware-dependent and is transparent to the application.

Z-bufferless HSR is performed if no z-buffer surface is attached to the rendering-target surface and the z-buffer comparison test is enabled (that is, when the state value associated with the **D3DRENDERSTATE_ZENABLE** enumeration constant is set to TRUE).

This flag was introduced in DirectX 5.

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*.)

D3DPBLENDCAP S_BOTHINVSRC ALPHA

Source blend factor is (1-As, 1-As, 1-As, 1-As) and destination blend factor is (As, As, As, As); the destination blend selection is overridden.

D3DPBLENDCAPS_BOTHSRCALPHA

Source blend factor is (As, As, As, As) and destination blend

factor is (1-As, 1-As, 1-As, 1-As); the destination blend selection is overridden.

D3DPBLENDCAPS_DESTALPHA

Blend factor is (Ad, Ad, Ad, Ad).

D3DPBLENDCAPS_DESTCOLOR

Blend factor is (Rd, Gd, Bd, Ad).

D3DPBLENDCAPS_INVDESTALPHA

Blend factor is (1-Ad, 1-Ad, 1-Ad, 1-Ad).

D3DPBLENDCAPS_INVDESTCOLOR

Blend factor is (1-Rd, 1-Gd, 1-Bd, 1-Ad).

D3DPBLENDCAPS_INVSRCALPHA

Blend factor is (1-As, 1-As, 1-As, 1-As).

D3DPBLENDCAPS_INVSRCCOLOR

Blend factor is (1-Rd, 1-Gd, 1-Bd, 1-Ad).

D3DPBLENDCAPS_ONE

Blend factor is (1, 1, 1, 1).

D3DPBLENDCAPS_SRCALPHA

Blend factor is (As, As, As, As).

D3DPBLENDCAPS_SRCALPHASAT

Blend factor is (f, f, f, 1); f = min(As, 1-Ad).

D3DPBLENDCAPS_SRCCOLOR

Blend factor is (Rs, Gs, Bs, As).

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. If this member is zero, the driver does not support alpha tests.

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).

With the monochromatic shade modes, the blue channel of the specular component is interpreted as a white intensity. (This is controlled by the **D3DRENDERSTATE_MONOENABLE** render state.)

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 currently supported.

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 currently supported.

D3DPSHADECAPS_FOGFLAT

D3DPSHADECAPS_FOGGOURAUD

D3DPSHADECAPS_FOGPHONG

Device can support fog in the flat, Gouraud, and Phong shading models, respectively. Phong shading is not currently supported.

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 currently supported.

dwTextureCaps

Miscellaneous texture-mapping capabilities. This member can be one or more of the following:

D3DPTEXTURECAPS_ALPHA

Supports RGBA textures 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

Supports texture mapping along borders.

D3DPTEXTURECAPS_PERSPECTIVE

Perspective correction is supported.

D3DPTEXTURECAPS_POW2

All nonmipmapped 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 2×2 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 D3DPTFILTERCAPS_MIPLINEAR, but interpolates between the two nearest mipmaps.

D3DPTFILTERCAPS_LINEARMIPNEAREST

The mipmap chosen is the mipmap whose texels most closely match the size of the pixel to be textured. The D3DFILTER_LINEAR method is then used with the texture.

D3DPTFILTERCAPS_MIPLINEAR

Two mipmaps are chosen whose texels most closely match the size of the pixel to be textured. The D3DFILTER_NEAREST method is then used with each texture to produce two values which are then weighted to produce a final texel value.

D3DPTFILTERCAPS_MIPNEAREST

Similar to D3DPTFILTERCAPS_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_ADD

Supports the additive texture-blending mode, in which the Gouraud interpolants are added to the texture lookup with saturation semantics. This capability corresponds to the **D3DTBLEND_ADD** member of the **D3DTEXTUREBLEND** enumerated type.

This flag was introduced in DirectX 5.

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:

D3DPTADDRESS CAPS_BORDER

Device supports setting coordinates outside the range [0.0, 1.0] to the border color, as specified by the **D3DRENDERSTATE_BORDERCOLOR** render state. This ability corresponds to the **D3DTADDRESS_BORDER** texture-addressing mode.

This flag was introduced in DirectX 5.

D3DPTADDRESSCAPS_CLAMP

Device can clamp textures to addresses.

D3DPTADDRESSCAPS_INDEPENDENTUV

Device can separate the texture-addressing modes of the U and V coordinates of the texture. This ability corresponds to the **D3DRENDERSTATE_TEXTUREADDRESSU** and **D3DRENDERSTATE_TEXTUREADDRESSV** render-state values.

This flag was introduced in DirectX 5.

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 32×32).

D3DPROCESSVERTICES

The **D3DPROCESSVERTICES** structure 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.

```
typedef struct _D3DPROCESSVERTICES {
    DWORD dwFlags;
    WORD  wStart;
    WORD  wDest;
    DWORD dwCount;
    DWORD dwReserved;
} D3DPROCESSVERTICES, *LPD3DPROCESSVERTICES;
```

Members

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.

See Also

D3DOPCODE

D3DRECT

The **D3DRECT** structure is a rectangle definition.

```
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;
```

Members

lX1 and **lY1**
Coordinates of the upper-left corner of the rectangle.

lX2 and **lY2**
Coordinates of the lower-right corner of the rectangle.

See Also

IDirect3DDevice::Pick, **IDirect3DViewport2::Clear**

D3DSPAN

The **D3DSPAN** structure 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.

```
typedef struct _D3DSPAN {
    WORD wCount;
    WORD wFirst;
} D3DSPAN, *LPD3DSPAN;
```

Members

wCount
Number of spans.

wFirst
Index to first vertex.

See Also

D3DOPCODE

D3DSTATE

The **D3DSTATE** structure 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.

```
typedef struct _D3DSTATE {
    union {
        D3DTRANSFORMSTATETYPE dtstTransformStateType;
        D3DLIGHTSTATETYPE     dlstLightStateType;
        D3DRENDERSTATETYPE    drstRenderStateType;
    };
    union {
        DWORD                 dwArg[1];
        D3DVALUE              dvArg[1];
    };
} D3DSTATE, *LPD3DSTATE;
```

Members

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.

See Also

D3DLIGHTSTATETYPE, **D3DOPCODE**, **D3DRENDERSTATETYPE**, and **D3DTRANSFORMSTATETYPE**, **D3DVALUE**

D3DSTATS

The **D3DSTATS** structure contains statistics used by the **IDirect3DDevice2::GetStats** method.

```
typedef struct _D3DSTATS {
    DWORD dwSize;
    DWORD dwTrianglesDrawn;
    DWORD dwLinesDrawn;
    DWORD dwPointsDrawn;
    DWORD dwSpansDrawn;
    DWORD dwVerticesProcessed;
} D3DSTATS, *LPD3DSTATS;
```

Members

dwSize
Size, in bytes, of this structure. This member must be initialized before the structure is used.

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.

See Also

IDirect3DDevice2::GetStats

D3DSTATUS

The **D3DSTATUS** structure 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.

```
typedef struct _D3DSTATUS {
    DWORD   dwFlags;
    DWORD   dwStatus;
    D3DRECT drExtent;
} D3DSTATUS, *LPD3DSTATUS;
```

Members

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_CLIPINTERSECTIONBACK

Logical **AND** of the clip flags for the vertices compared to the back clipping plane 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_CLIPINTERSECTIONGEN0 through D3DSTATUS_CLIPINTERSECTIONGEN5

Logical **AND** of the clip flags for application-defined clipping planes.

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.

Clip Union Flags

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

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.

Remarks

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.

See Also

D3DEXECUTEDATA, **D3DOPCODE**, **D3DRECT**

D3DTEXTURELOAD

The **D3DTEXTURELOAD** structure describes operand data for the **D3DOP_TEXTURELOAD** opcode in the **D3DOPCODE** enumerated type.

```
typedef struct _D3DTEXTURELOAD {
    D3DTEXTUREHANDLE hDestTexture;
    D3DTEXTUREHANDLE hSrcTexture;
} D3DTEXTURELOAD, *LPD3DTEXTURELOAD;
```

Members

hDestTexture
Handle to the destination texture.

hSrcTexture
Handle to the source texture.

Remarks

The textures referred to by the **hDestTexture** and **hSrcTexture** members must be the same size.

D3DTLVERTEX

The **D3DTLVERTEX** structure defines a transformed and lit vertex (screen coordinates with color) for the **D3DLIGHTDATA** structure.

```
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;
```

Members

dvSX, **dvSY**, and **dvSZ**
Values of the **D3DVALUE** type describing a vertex in screen coordinates. The largest allowable value for **dvSZ** is 0.99999, if you want the vertex to be within the range of z-values that are displayed.

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.

Remarks

Direct3D uses the current viewport parameters (the **dwX**, **dwY**, **dwWidth**, and **dwHeight** members of the **D3DVIEWPORT2** structure) to clip **D3DTLVERTEX** vertices. The system always clips z coordinates to [0, 1]. To prevent the system from clipping these vertices, use the D3DDP_DONOTCLIP flag in your call to **IDirect3DDevice2::Begin**.

Prior to DirectX 5, Direct3D did not clip **D3DTLVERTEX** vertices.

See Also

D3DLIGHTDATA, **D3DLVERTEX**, **D3DVERTEX**

D3DTRANSFORMCAPS

The **D3DTRANSFORMCAPS** structure describes the transformation capabilities of a device. This structure is part of the **D3DDEVICEDESC** structure.

```
typedef struct _D3DTransformCaps {
    DWORD dwSize;
    DWORD dwCaps;
} D3DTRANSFORMCAPS, *LPD3DTRANSFORMCAPS;
```

Members

dwSize

Size, in bytes, of this structure. This member must be initialized before the structure is used.

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

The **D3DTRANSFORMDATA** structure contains information about transformations for the **IDirect3DViewport2::TransformVertices** method.

```
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;
```

Members

dwSize
Size of the structure, in bytes. This member must be initialized before the structure is used.

lpIn
Address of the vertices to be transformed. This should be a **D3DLVERTEX** structure.

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_GEN0 through D3DCLIP_GEN5
Application-defined clipping planes.

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.

dwClipIntersection

Flags denoting the intersection of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPINTERSECTIONBACK

Logical **AND** of the clip flags for the vertices compared to the back clipping plane 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_CLIPINTERSECTIONGEN0 through D3DSTATUS_CLIPINTERSECTIONGEN5

Logical **AND** of the clip flags for application-defined clipping planes.

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.

dwClipUnion

Flags denoting the union of the clip flags. This member can be one or more of the following values:

D3DSTATUS_CLIPUNIONBACK

Equal to D3DCLIP_BACK.

D3DSTATUS_CLIPUNIONBOTTOM

Equal to D3DCLIP_BOTTOM.

D3DSTATUS_CLIPUNIONFRONT

Equal to D3DCLIP_FRONT.

D3DSTATUS_CLIPUNIONGEN0 through D3DSTATUS_CLIPUNIONGEN5

Equal to D3DCLIP_GEN0 through D3DCLIP_GEN5.

D3DSTATUS_CLIPUNIONLEFT

Equal to D3DCLIP_LEFT.

D3DSTATUS_CLIPUNIONRIGHT

Equal to D3DCLIP_RIGHT.

D3DSTATUS_CLIPUNIONTOP

Equal to D3DCLIP_TOP.

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

Remarks

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.

See Also

IDirect3DViewport2::TransformVertices

D3DTRIANGLE

The **D3DTRIANGLE** structure 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.

```
typedef struct _D3DTRIANGLE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
    union {
        WORD v3;
        WORD wV3;
    };
    WORD     wFlags;
} D3DTRIANGLE, *LPD3DTRIANGLE;
```

Members

wV1, **wV2**, and **wV3**

Vertices describing the triangle.

wFlags

This value can be a combination of the following flags:

Edge flags

These flags describe which edges of the triangle to enable. (This information is useful only in wireframe mode.)

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)

Cull or render the triangles in the strip or fan based on the treatment of this triangle. That is, if this triangle is culled, also cull the specified number of subsequent triangles. If this triangle is rendered, also render the specified number of subsequent triangles.

This length must be greater than zero and less than 30.

Remarks

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

For more information, see Triangle Strips and Fans.

D3DVECTOR

The **D3DVECTOR** structure defines a vector for many Direct3D and Direct3DRM methods and structures.

```
typedef struct _D3DVECTOR {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
} D3DVECTOR, *LPD3DVECTOR;
```

Members

dvX, **dvY**, and **dvZ**

Values of the **D3DVALUE** type describing the vector.

See Also

D3DLIGHT2, **D3DLIGHTINGELEMENT**,

D3DVERTEX

The **D3DVERTEX** structure 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.

```
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;
```

Members

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.

See Also

D3DLVERTEX, **D3DTLVERTEX**, **D3DVALUE**,

D3DVIEWPORT

The **D3DVIEWPORT** structure defines the visible 3-D volume and the 2-D screen area that a 3-D volume projects onto for the **IDirect3DViewport2::GetViewport** and **IDirect3DViewport2::SetViewport** methods.

For the **IDirect3D2** and **IDirect3DDevice2** interfaces, this structure has been superseded by the **D3DVIEWPORT2** structure.

```
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;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

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 how coordinates are scaled. The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

dvMaxX, **dvMaxY**, **dvMinZ**, and **dvMaxZ**
Values of the **D3DVALUE** type describing the maximum and minimum nonhomogeneous coordinates of x, y, and z. Again, the relevant coordinates are the nonhomogeneous coordinates that result from the perspective division.

Remarks

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.

See Also

D3DVALUE, **IDirect3DViewport2::GetViewport**, **IDirect3DViewport2::SetViewport**

D3DVIEWPORT2

The **D3DVIEWPORT2** structure defines the visible 3-D volume and the window dimensions that a 3-D volume projects onto. This structure is used by the methods of the **IDirect3D2** and **IDirect3DDevice2** interfaces, and in particular by the **IDirect3DViewport2::GetViewport2** and **IDirect3DViewport2::SetViewport2** methods. This structure was introduced in DirectX 5.

```
typedef struct _D3DVIEWPORT2 {
    DWORD       dwSize;
    DWORD       dwX;
    DWORD       dwY;
    DWORD       dwWidth;
    DWORD       dwHeight;
    D3DVALUE    dvClipX;
    D3DVALUE    dvClipY;
    D3DVALUE    dvClipWidth;
    D3DVALUE    dvClipHeight;
    D3DVALUE    dvMinZ;
    D3DVALUE    dvMaxZ;
} D3DVIEWPORT2, *LPD3DVIEWPORT2;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

dwX and **dwY**
Coordinates of the top-left corner of the viewport. Unless you want to render to a subset of the surface, these members can be set to 0.

dwWidth and **dwHeight**
Dimensions of the viewport.

dvClipX and **dvClipY**
Coordinates of the top-left corner of the clipping volume.

The relevant coordinates here are the nonhomogeneous coordinates that result from the perspective division that projects the vertices onto the w=1 plane.

dvClipWidth and **dvClipHeight**
Dimensions of the clipping volume projected onto the w=1 plane. Unless you want to render to a subset of the surface, these members can be set to the width and height of the destination surface.

dvMinZ and **dvMaxZ**
Values of the **D3DVALUE** type describing the maximum and minimum nonhomogeneous z-coordinates resulting from the perspective divide and projected onto the w=1 plane.

Remarks

The coordinates and dimensions of the viewport are given relative to the top left of the device; values increase in the y-direction as you descend the screen.

If you are using **D3DVERTEX** or **D3DLVERTEX** vertices – that is, if Direct3D is performing the transformations – you might want to set the last six members of this structure as follows:

```
float inv_aspect = (float)dwHeight/dwWidth;

dvClipX = -1.0f;
dvClipY = inv_aspect;
dvClipWidth = 2.0f;
dvClipHeight = 2.0f * inv_aspect;
```

```
dvMinZ = 0.0f;
dvMaxZ = 1.0f;
```

By taking the aspect ratio into account you are assured that as the surface is resized the angle of the horizontal field of view remains constant. This prevents unexpected distortions when the user pulls the window into an unusual shape. If distortion is not an issue in your application, set *aspect* to 1. Notice that dividing the height by the width produces an inverse aspect ratio; in Direct3D, the aspect ratio is defined by dividing the width by the height.

If you are using **D3DTLVERTEX** vertices – that is, if your application is taking care of the transformations and lighting – you can set up the clip space however is best for your application. If the x- and y-coordinates in your data already match pixels, you could set the last six members of **D3DVIEWPORT2** as follows:

```
dvClipX = 0;
dvClipY = 0;
dvClipWidth = dwWidth;
dvClipHeight = dwHeight;
dvMinZ = 0.0f;
dvMaxZ = 1.0f;
```

Unlike the **D3DVIEWPORT** structure, **D3DVIEWPORT2** specifies the relationship between the size of the viewport and the window.

When the viewport is changed, the driver builds a new transformation matrix.

For more information about working with viewports, see Viewports and Transformations.

See Also

D3DVALUE, **IDirect3DViewport2::GetViewport2**, **IDirect3DViewport2::SetViewport2**

Enumerated Types

This section contains information about the following enumerated types used with Direct3D Immediate Mode.

- **D3DANTIALIASMODE**
- **D3DBLEND**
- **D3DCMPFUNC**
- **D3DCOLORMODEL**
- **D3DCULL**
- **D3DFILLMODE**
- **D3DFOGMODE**
- **D3DLIGHTSTATETYPE**
- **D3DLIGHTTYPE**
- **D3DOPCODE**
- **D3DPRIMITIVETYPE**
- **D3DRENDERSTATETYPE**
- **D3DSHADEMODE**
- **D3DTEXTUREADDRESS**
- **D3DTEXTUREBLEND**
- **D3DTEXTUREFILTER**
- **D3DTRANSFORMSTATETYPE**
- **D3DVERTEXTYPE**

D3DANTIALIASMODE

The **D3DANTIALIASMODE** enumerated type defines the supported antialiasing mode for the **D3DRENDERSTATE_ANTIALIAS** value in the **D3DRENDERSTATETYPE** enumerated type. These values define the settings for antialiasing the edges of primitives.

This type was introduced with DirectX 5.

```
typedef enum _D3DANTIALIASMODE {
    D3DANTIALIAS_NONE          = 0,
    D3DANTIALIAS_SORTDEPENDENT = 1,
    D3DANTIALIAS_SORTINDEPENDENT = 2
    D3DANTIALIAS_FORCE_DWORD     = 0x7fffffff,
} D3DANTIALIASMODE;
```

Members

D3DANTIALIAS_NONE
No antialiasing is performed. This is the default setting.

D3DANTIALIAS_SORTDEPENDENT
Antialiasing is dependent on the sort order of the polygons (back-to-front or front-to-back). The application must draw polygons in the right order for antialiasing to occur.

D3DANTIALIAS_SORTINDEPENDENT
Antialiasing is not dependent on the sort order of the polygons.

D3DANTIALIAS_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

D3DBLEND

The **D3DBLEND** enumerated type defines the supported blend mode for the **D3DRENDERSTATE_DESTBLEND** 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*.

```
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_FORCE_DWORD   = 0x7fffffff,
} D3DBLEND;
```

Members

D3DBLEND_ZERO
Blend factor is (0, 0, 0, 0).

D3DBLEND_ONE
Blend factor is (1, 1, 1, 1).

D3DBLEND_SRCCOLOR
Blend factor is (Rs, Gs, Bs, As).

D3DBLEND_INVSRCCOLOR
Blend factor is (1-Rs, 1-Gs, 1-Bs, 1-As).

D3DBLEND_SRCALPHA
Blend factor is (As, As, As, As).

D3DBLEND_INVSRCALPHA
Blend factor is (1-As, 1-As, 1-As, 1-As).

D3DBLEND_DESTALPHA
Blend factor is (Ad, Ad, Ad, Ad).

D3DBLEND_INVDESTALPHA
Blend factor is (1-Ad, 1-Ad, 1-Ad, 1-Ad).

D3DBLEND_DESTCOLOR
Blend factor is (Rd, Gd, Bd, Ad).

D3DBLEND_INVDESTCOLOR
Blend factor is (1-Rd, 1-Gd, 1-Bd, 1-Ad).

D3DBLEND_SRCALPHASAT
Blend factor is (f, f, f, 1); f = min(As, 1-Ad).

D3DBLEND_BOTHSRCALPHA
Source blend factor is (As, As, As, As), and destination blend factor is (1-As, 1-As, 1-As, 1-As); the destination blend selection is overridden.

D3DBLEND_BOTHINVSRCALPHA

Source blend factor is (1-As, 1-As, 1-As, 1-As), and destination blend factor is (As, As, As, As); the destination blend selection is overridden.

D3DBLEND_FORCE_DWORD

Forces this enumerated type to be 32 bits in size.

D3DCMPFUNC

The **D3DCMPFUNC** enumerated type defines the supported compare functions for the **D3DRENDERSTATE_ZFUNC** and **D3DRENDERSTATE_ALPHAFUNC** values of the **D3DRENDERSTATETYPE** enumerated type.

```
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,
    D3DCMP_FORCE_DWORD   = 0x7fffffff,
} D3DCMPFUNC;
```

Members

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.

D3DCMP_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

D3DCOLORMODEL

The **D3DCOLORMODEL** constant defines the color model in which the system will run. A driver can expose either or both flags in the **dcmColorModel** member of the **D3DDEVICEDESC** structure.

```
typedef DWORD D3DCOLORMODEL
```

Values

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.

Remarks

Prior to DirectX 5, these values were part of an enumerated type. This was not correct, because they are bit flags. The enumerated type in earlier versions of DirectX had this syntax:

```
typedef enum _D3DCOLORMODEL {
    D3DCOLOR_MONO = 1,
    D3DCOLOR_RGB  = 2,
} D3DCOLORMODEL;
```

See Also

D3DDEVICEDESC, **D3DFINDDEVICESEARCH**, **D3DLIGHTSTATETYPE**

D3DCULL

The **D3DCULL** enumerated type defines the supported cull modes. These define how back faces are culled when rendering a geometry.

```
typedef enum _D3DCULL {
    D3DCULL_NONE = 1,
    D3DCULL_CW   = 2,
    D3DCULL_CCW  = 3,
    D3DCULL_FORCE_DWORD   = 0x7fffffff,
} D3DCULL;
```

Members

D3DCULL_NONE
Do not cull back faces.

D3DCULL_CW
Cull back faces with clockwise vertices.

D3DCULL_CCW
Cull back faces with counterclockwise vertices.

D3DCULL_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

D3DPRIMCAPS, **D3DRENDERSTATETYPE**

D3DFILLMODE

The **D3DFILLMODE** enumerated type contains constants describing the fill mode. These values are used by the **D3DRENDERSTATE_FILLMODE** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFILLMODE {
    D3DFILL_POINT     = 1,
    D3DFILL_WIREFRAME = 2,
    D3DFILL_SOLID     = 3
    D3DFILL_FORCE_DWORD   = 0x7fffffff,
} D3DFILLMODE;
```

Members

D3DFILL_POINT
Fill points.

D3DFILL_WIREFRAME
Fill wireframes. This fill mode currently does not work for clipped primitives when you are using the DrawPrimitive methods.

D3DFILL_SOLID
Fill solids.

D3DFILL_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

D3DFOGMODE

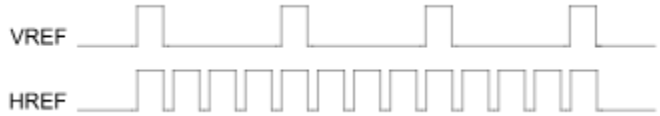
The **D3DFOGMODE** enumerated type contains constants describing the fog mode. These values are used by the **D3DRENDERSTATE_FOGTABLEMODE** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE   = 0,
    D3DFOG_EXP    = 1,
    D3DFOG_EXP2   = 2,
    D3DFOG_LINEAR = 3
    D3DFOG_FORCE_DWORD   = 0x7fffffff,
} D3DFOGMODE;
```

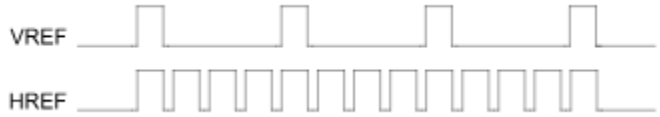
Members

D3DFOG_NONE
No fog effect.

D3DFOG_EXP
The fog effect intensifies exponentially, according to the following formula:



D3DFOG_EXP2
The fog effect intensifies exponentially with the square of the distance, according to the following formula:



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 currently supported.

D3DFOG_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color will work, since in this case any fog color is effectively black.)

For more information about fog, see Colors and Fog.

Note 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

The **D3DLIGHTSTATETYPE** enumerated type defines the light state for the **D3DOP_STATELIGHT** opcode. This enumerated type is part of the **D3DSTATE** structure.

```
typedef enum _D3DLIGHTSTATETYPE {
    D3DLIGHTSTATE_MATERIAL   = 1,
    D3DLIGHTSTATE_AMBIENT    = 2,
    D3DLIGHTSTATE_COLORMODEL = 3,
    D3DLIGHTSTATE_FOGMODE    = 4,
    D3DLIGHTSTATE_FOGSTART   = 5,
    D3DLIGHTSTATE_FOGEND     = 6,
    D3DLIGHTSTATE_FOGDENSITY = 7,
    D3DLIGHTSTATE_FORCE_DWORD     = 0x7fffffff,
} D3DLIGHTSTATETYPE;
```

Members

D3DLIGHTSTATE_MATERIAL
Defines the material that is lit and used to compute the final color and intensity values during rasterization. The default value is NULL.

This value must be set when you use textures in ramp mode.

D3DLIGHTSTATE_AMBIENT
Sets the color and intensity of the current ambient light. If an application specifies this value, it should not specify a light as a parameter. 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.

D3DLIGHTSTATE_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

D3DOPCODE and **D3DSTATE**

D3DLIGHTTYPE

The **D3DLIGHTTYPE** enumerated type defines the light type. This enumerated type is part of the **D3DLIGHT2** structure.

```
typedef enum _D3DLIGHTTYPE {
    D3DLIGHT_POINT         = 1,
    D3DLIGHT_SPOT          = 2,
    D3DLIGHT_DIRECTIONAL   = 3,
    D3DLIGHT_PARALLELPOINT = 4,
    D3DLIGHT_FORCE_DWORD   = 0x7fffffff,
} D3DLIGHTTYPE;
```

Members

D3DLIGHT_POINT
Light is a point source. The light has a position in space and radiates light in all directions.

D3DLIGHT_SPOT
Light is a spotlight source. This light is something like a point light except that the illumination is limited to a cone. This light type has a direction and several other parameters which determine the shape of the cone it produces. For information about these parameters, see the **D3DLIGHT2** structure.

D3DLIGHT_DIRECTIONAL
Light is a directional source. This is equivalent to using a point light source at an infinite distance.

D3DLIGHT_PARALLELPOINT
Light is a parallel point source. This light type acts like a directional light except its direction is the vector going from the light position to the origin of the geometry it is illuminating.

D3DLIGHT_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

Directional and parallel-point lights are slightly faster than point light sources, but point lights look a little better. Spotlights offer interesting visual effects but are computationally expensive.

D3DOPCODE

The **D3DOPCODE** enumerated type contains the opcodes for execute buffer.

```
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,
    D3DOP_FORCE_DWORD     = 0x7fffffff,
} D3DOPCODE;
```

Members

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.

D3DOP_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

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.

See Also

D3DINSTRUCTION

D3DPRIMITIVETYPE

The **D3DPRIMITIVETYPE** enumerated type lists the primitives supported by DrawPrimitive methods. This type was introduced in DirectX 5.

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST     = 1,
    D3DPT_LINELIST      = 2,
    D3DPT_LINESTRIP     = 3,
    D3DPT_TRIANGLELIST  = 4,
    D3DPT_TRIANGLESTRIP = 5,
    D3DPT_TRIANGLEFAN   = 6
    D3DPT_FORCE_DWORD   = 0x7fffffff,
} D3DPRIMITIVETYPE;
```

Members

D3DPT_POINTLIST
Renders the vertices as a collection of isolated points.

D3DPT_LINELIST
Renders the vertices as a list of isolated straight line segments. Calls using this primitive type will fail if the count is less than 2, or is odd.

D3DPT_LINESTRIP
Renders the vertices as a single polyline. Calls using this primitive type will fail if the count is less than 2.

D3DPT_TRIANGLELIST
Renders the specified vertices as a sequence of isolated triangles. Each group of 3 vertices defines a separate triangle. Calls using this primitive type will fail if the count is less than 3, or if not evenly divisible by 3.

Backface culling is affected by the current winding order render state.

D3DPT_TRIANGLESTRIP
Renders the vertices as a triangle strip. Calls using this primitive type will fail if the count is less than 3.

The backface removal flag is automatically flipped on even numbered triangles.

D3DPT_TRIANGLEFAN
Renders the vertices as a triangle fan. Calls using this primitive type will fail if the count is less than 3.

D3DPT_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

IDirect3DDevice2::Begin, **IDirect3DDevice2::BeginIndexed**, **IDirect3DDevice2::DrawIndexedPrimitive**, **IDirect3DDevice2::DrawPrimitive**

D3DRENDERSTATETYPE

The **D3DRENDERSTATETYPE** enumerated type 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.

Values 40 through 49 were introduced with DirectX 5.

```
typedef enum _D3DRENDERSTATETYPE {
    D3DRENDERSTATE_TEXTUREHANDLE      = 1,    // texture handle
    D3DRENDERSTATE_ANTIALIAS          = 2,    // antialiasing mode
    D3DRENDERSTATE_TEXTUREADDRESS     = 3,    // texture address
    D3DRENDERSTATE_TEXTUREPERSPECTIVE = 4,    // perspective correction
    D3DRENDERSTATE_WRAPU              = 5,    // wrap in u direction
    D3DRENDERSTATE_WRAPV              = 6,    // wrap in v direction
    D3DRENDERSTATE_ZENABLE            = 7,    // enable z test
    D3DRENDERSTATE_FILLMODE           = 8,    // fill mode
    D3DRENDERSTATE_SHADEMODE          = 9,    // shade mode
    D3DRENDERSTATE_LINEPATTERN        = 10,   // line pattern
    D3DRENDERSTATE_MONOENABLE         = 11,   // enable mono rendering
    D3DRENDERSTATE_ROP2               = 12,   // raster operation
    D3DRENDERSTATE_PLANEMASK          = 13,   // physical plane mask
    D3DRENDERSTATE_ZWRITEENABLE       = 14,   // enable z writes
    D3DRENDERSTATE_ALPHATESTENABLE    = 15,   // enable alpha tests
    D3DRENDERSTATE_LASTPIXEL          = 16,   // draw last pixel in a line
    D3DRENDERSTATE_TEXTUREMAG         = 17,   // how textures are magnified
    D3DRENDERSTATE_TEXTUREMIN         = 18,   // how textures are reduced
    D3DRENDERSTATE_SRCBLEND           = 19,   // blend factor for source
    D3DRENDERSTATE_DESTBLEND          = 20,   // blend factor for
destination
    D3DRENDERSTATE_TEXTUREMAPBLEND    = 21,   // blend mode for map
    D3DRENDERSTATE_CULLMODE           = 22,   // back-face culling mode
    D3DRENDERSTATE_ZFUNC              = 23,   // z-comparison function
    D3DRENDERSTATE_ALPHAREF           = 24,   // reference alpha value
    D3DRENDERSTATE_ALPHAFUNC          = 25,   // alpha-comparison function
    D3DRENDERSTATE_DITHERENABLE       = 26,   // enable dithering
    D3DRENDERSTATE_BLENDENABLE        = 27,   // replaced by
D3DRENDERSTATE_ALPHABLENDENABLE
    D3DRENDERSTATE_FOGENABLE          = 28,   // enable fog
    D3DRENDERSTATE_SPECULARENABLE     = 29,   // enable specular highlights
    D3DRENDERSTATE_ZVISIBLE           = 30,   // enable z-checking
    D3DRENDERSTATE_SUBPIXEL           = 31,   // enable subpixel correction
    D3DRENDERSTATE_SUBPIXELX          = 32,   // enable x subpixel
correction
    D3DRENDERSTATE_STIPPLEDALPHA      = 33,   // enable stippled alpha
    D3DRENDERSTATE_FOGCOLOR           = 34,   // fog color
    D3DRENDERSTATE_FOGTABLEMODE       = 35,   // fog mode
    D3DRENDERSTATE_FOGTABLESTART      = 36,   // fog table start
    D3DRENDERSTATE_FOGTABLEEND        = 37,   // fog table end
    D3DRENDERSTATE_FOGTABLEDENSITY    = 38,   // fog density
    D3DRENDERSTATE_STIPPLEENABLE      = 39,   // enables stippling
    D3DRENDERSTATE_EDGEANTIALIAS      = 40,   // antialias edges
    D3DRENDERSTATE_COLORKEYENABLE     = 41,   // enable color-key
transparency
```

```
    D3DRENDERSTATE_ALPHABLENDENABLE   = 42,   // enable alpha-blend
transparency
    D3DRENDERSTATE_BORDERCOLOR        = 43,   // border color
    D3DRENDERSTATE_TEXTUREADDRESSU    = 44,   // u texture address mode
    D3DRENDERSTATE_TEXTUREADDRESSV    = 45,   // v texture address mode
    D3DRENDERSTATE_MIPMAPLODBIAS      = 46,   // mipmap LOD bias
    D3DRENDERSTATE_ZBIAS              = 47,   // z bias
    D3DRENDERSTATE_RANGEFOGENABLE     = 48,   // enables range-based fog
    D3DRENDERSTATE_ANISOTROPY         = 49,   // max. anisotropy
    D3DRENDERSTATE_STIPPLEPATTERN00   = 64,   // first line of stipple
pattern
        // Stipple patterns 01 through 30 omitted here.
    D3DRENDERSTATE_STIPPLEPATTERN31   = 95,   // last line of stipple
pattern
    D3DRENDERSTATE_FORCE_DWORD        = 0x7fffffff,
} D3DRENDERSTATETYPE;
```

Members

D3DRENDERSTATE_TEXTUREHANDLE

Texture handle. The default value is NULL, which disables texture mapping and reverts to flat or Gouraud shading.

If the specified texture is in a system memory surface and the driver can only support texturing from display memory surfaces, the call will fail.

In retail builds the texture handle is not validated.

D3DRENDERSTATE_ANTIALIAS

One of the members of the **D3DANTIALIASMODE** enumerated type specifying the antialiasing of primitive edges. The default value is **D3DANTIALIAS_NONE**.

D3DRENDERSTATE_TEXTUREADDRESS

One of the members of the **D3DTEXTUREADDRESS** enumerated type. The default value is **D3DTADDRESS_WRAP**.

Applications that need to specify separate texture-addressing modes for the U and V coordinates of a texture can use the **D3DRENDERSTATE_TEXTUREADDRESSU** and **D3DRENDERSTATE_TEXTUREADDRESSV** render states.

D3DRENDERSTATE_TEXTUREPERSPECTIVE

TRUE for perspective correction. The default value is FALSE.

If a square were exactly perpendicular to the viewer, all the points in the square would appear the same. But if the square were tilted with respect to the viewer so that one edge was closer than the other, one side would appear to be longer than the other. Perspective correction ensures that the interpolation of texture coordinates happens correctly in such cases.

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_MONOENABLE

TRUE to enable monochromatic rendering, using a grayscale based on the blue channel of the color rather than full RGB. The default value is FALSE. If the device does not support RGB rendering, the value will be TRUE. Applications can check whether the device supports RGB rendering by using the **dcmColorModel** member of the **D3DDEVICEDESC** structure.

In monochromatic rendering, only the intensity (grayscale) component of the color and specular components are interpolated across the triangle. This means that only one channel (gray) is interpolated across the triangle instead of 3 channels (R,G,B), which is a performance gain for some hardware. This grayscale component is derived from the blue channel of the color and specular components of the triangle.

D3DRENDERSTATE_ROP2

One of the 16 standard Windows 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 the bitwise negation of zero (~0). This physical plane mask can be used to turn off the red bit, the blue bit, and so on.

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. If this state is FALSE, z comparisons are still made according to the render state **D3DRENDERSTATE_ZFUNC** (assuming z-buffering is taking place), but z values are not written to the z-buffer.

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.

The incoming alpha value is compared with the reference alpha value using the comparison function provided by the **D3DRENDERSTATE_ALPHAFUNC** render state. When this mode is enabled, alpha blending occurs only if the test succeeds.

D3DRENDERSTATE_LASTPIXEL

TRUE to prevent drawing the last pixel in a line or triangle. The default value is FALSE.

D3DRENDERSTATE_TEXTUREMAG

One of the members of the **D3DTEXTUREFILTER** enumerated type. This render state describes how a texture should be filtered when it is being magnified (that is, when a texel must cover more than one pixel). The valid values are **D3DFILTER_NEAREST** (the default) and **D3DFILTER_LINEAR**.

D3DRENDERSTATE_TEXTUREMIN

One of the members of the **D3DTEXTUREFILTER** enumerated type. This render state describes how a texture should be filtered when it is being made smaller (that is, when a pixel contains more than one texel). Any of the members of the **D3DTEXTUREFILTER** enumerated type can be specified for this render state. The default value is **D3DFILTER_NEAREST**.

D3DRENDERSTATE_SRCBLEND

One of the members of the **D3DBLEND** enumerated type. The default value is D3DBLEND_ONE.

D3DRENDERSTATE_DESTBLEND

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.

The z value of the pixel is compared with the z-buffer value. If the z value of the pixel passes the comparison function, the pixel is written.

The z value is written to the z-buffer only if the render state **D3DRENDERSTATE_ZWRITEENABLE** is TRUE.

Software rasterizers and many hardware accelerators work faster if the z test fails, since there is no need to filter and modulate the texture if the pixel is not going to be rendered.

D3DRENDERSTATE_ALPHAREF

Value specifying a reference alpha value against which pixels are tested when alpha-testing is enabled. This value's type is **D3DFIXED**. It is a 16.16 fixed-point value in the range [0 - 1]. 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

Replaced by the **D3DRENDERSTATE_ALPHABLENDENABLE** render state for DirectX 5.

D3DRENDERSTATE_FOGENABLE

TRUE to enable fog. The default value is FALSE.

D3DRENDERSTATE_SPECULARENABLE

TRUE to enable specular highlights. The default value is TRUE.

Specular highlights are calculated as though every vertex in the object being lit were at the object's origin. This gives the expected results as long as the object is modeled around the origin and the distance from the light to the object is relatively large.

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.

In this mode of operation, the primitives are rendered without writing pixels or updating the z-buffer, and the driver returns TRUE if any of them would be visible. Since no pixels are rendered, this operation is often much faster than it would be if the primitives were naively rendered.

Direct3D's retained mode uses this operation as a quick-reject test: it does the z-visible test on the bounding box of a set of primitives and only renders them if it returns TRUE.

D3DRENDERSTATE_SUBPIXEL

TRUE to enable subpixel correction. The default value is FALSE.

Subpixel correction is the ability to draw pixels in precisely their correct locations. In a system that implemented subpixel correction, if a pixel were at position 0.1356, its position would be interpolated from the actual coordinate rather than simply drawn at 0 (using the integer values). Hardware can be non subpixel correct or subpixel correct in x or in both x and y. When interpolating across the x-direction the actual coordinate is used. All hardware should be subpixel correct. Some software rasterizers are not subpixel correct because of the performance loss.

Subpixel correction means that the hardware always pre-steps the interpolant values in the x-direction to the nearest pixel centers and then steps one pixel at a time in the y-direction. For each x span it also pre-steps in the x-direction to the nearest pixel center and then steps in the x-direction one pixel each time. This results in very accurate rendering and eliminates almost all jittering of pixels on triangle edges. Most hardware either doesn't support it (always off) or always supports it (always on).

D3DRENDERSTATE_SUBPIXELX

TRUE to enable subpixel correction in the x direction only. The default value is FALSE.

D3DRENDERSTATE_STIPPLEDALPHA

TRUE to enable stippled alpha. The default value is FALSE.

Current software rasterizers ignore this render state. You can use the D3DPSHADECAPS_ALPHAFLATSTIPPLED flag in the **D3DPRIMCAPS** structure to discover whether the current hardware supports this render state.

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

Position in fog table at which fog effects begin for linear fog mode. You specify a position in the fog table with a value between 0.0 and 1.0. This render state enables you to exclude fog effects for positions close to the camera; for example, you could set this value to 0.3 to prevent fog effects for positions between 0.0 and 0.299.

D3DRENDERSTATE_FOGTABLEEND

Position in fog table at which fog effects end for linear fog mode. You specify a position in the fog table with a value between 0.0 and 1.0. This render state enables you to set a position in the fog table at which fog effects will not increase. For example, you could set this value to 0.7 to prevent additional fog effects for positions between 0.701 and 1.0.

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 overrides the current stipple pattern, as specified by the **D3DRENDERSTATE_STIPPLEPATTERN00** through **D3DRENDERSTATE_STIPPLEPATTERN31** render states. When stippled alpha is disabled, the stipple pattern must be returned.

D3DRENDERSTATE_EDGEANTIALIAS

TRUE to antialias lines forming the convex outline of objects. The default value is FALSE. When set to TRUE, only lines should be drawn. The behavior is undefined if triangles or points are drawn when this render state is set. Antialiasing is performed simply by averaging the values of neighboring pixels. Although this is not the best way to perform antialiasing, it can be very efficient; hardware that supports this kind of operation is becoming more common.

Applications should not antialias interior edges of objects. The lines forming the outside edges should be drawn last.

D3DRENDERSTATE_COLORKEYENABLE

TRUE to enable color-keyed transparency. The default value is FALSE. You can use this render state with **D3DRENDERSTATE_ALPHABLENDENABLE** to implement fine blending control.

This render state was introduced in DirectX 5. Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** structure to find out whether this render state is supported.

D3DRENDERSTATE_ALPHABLENDENABLE

TRUE to enable alpha-blended transparency. The default value is FALSE.

Prior to DirectX 5, this render state was called **D3DRENDERSTATE_BLENDENABLE**. Its name was changed to make its meaning more explicit.

Prior to DirectX 5, the software rasterizers used this render state to toggle both color keying and alpha blending. With DirectX 5, you should use the **D3DRENDERSTATE_COLORKEYENABLE** render state to toggle color keying. (Hardware rasterizers have always used the **D3DRENDERSTATE_BLENDENABLE** render state only for toggling alpha blending.)

The type of alpha blending is determined by the **D3DRENDERSTATE_SRCBLEND** and **D3DRENDERSTATE_DESTBLEND** render states. **D3DRENDERSTATE_ALPHABLENDENABLE**, with **D3DRENDERSTATE_COLORKEYENABLE**, allows fine blending control.

D3DRENDERSTATE_ALPHABLENDENABLE does not affect the texture-blending modes specified by the **D3DTEXTUREBLEND** enumerated type. Texture blending is logically well before the **D3DRENDERSTATE_ALPHABLENDENABLE** part of the pixel pipeline. The only interaction between the two is that the alpha portions remaining in the polygon after the **D3DTEXTUREBLEND** phase may be used in the **D3DRENDERSTATE_ALPHABLENDENABLE** phase to govern interaction with the content in the frame buffer.

Applications should check the D3DDEVCAPS_DRAWPRIMTLVERTEX flag in the **D3DDEVICEDESC** structure to find out whether this render state is supported.

D3DRENDERSTATE_BORDERCOLOR

A **DWORD** value specifying a border color. If the texture addressing mode is specified as **D3DTADDRESS_BORDER** (as set in the **D3DTEXTUREADDRESS** enumerated type), this render state specifies the border color the system uses when it encounters texture coordinates outside the range [0.0, 1.0].

The format of the physical-color information specified by the **DWORD** value depends on the format of the DirectDraw surface.

D3DRENDERSTATE_TEXTUREADDRESSU

One of the members of the **D3DTEXTUREADDRESS** enumerated type. The default value is **D3DTADDRESS_WRAP**. This render state applies only to the U texture coordinate.

This render state, along with **D3DRENDERSTATE_TEXTUREADDRESSV**, allows you to specify separate texture-addressing modes for the U and V coordinates of a texture. Because the **D3DRENDERSTATE_TEXTUREADDRESS** render state applies to both the U and V texture coordinates, it overrides any values set for the **D3DRENDERSTATE_TEXTUREADDRESSU** render state.

D3DRENDERSTATE_TEXTUREADDRESSV

One of the members of the **D3DTEXTUREADDRESS** enumerated type. The default value is **D3DTADDRESS_WRAP**. This render state applies only to the V texture coordinate.

This render state, along with **D3DRENDERSTATE_TEXTUREADDRESSU**, allows you to specify separate texture-addressing modes for the U and V coordinates of a texture. Because the **D3DRENDERSTATE_TEXTUREADDRESS** render state applies to both the U and V texture coordinates, it overrides any values set for the **D3DRENDERSTATE_TEXTUREADDRESSV** render state.

D3DRENDERSTATE_MIPMAPLODBIAS

Floating-point **D3DVALUE** value used to change the level of detail (LOD) bias. This value offsets the value of the mipmap level that is computed by trilinear texturing. It is usually in the range -1.0 to 1.0; the default value is 0.0.

Each unit bias (+/-1.0) biases the selection by exactly one mipmap level. A positive bias will cause the use of larger mipmap levels, resulting in a sharper but more aliased image. A negative bias will cause the use of smaller mipmap levels, resulting in a blurrier image. Applying a negative bias also results in the referencing of a smaller amount of texture data, which can boost performance on some systems.

D3DRENDERSTATE_ZBIAS

An integer value in the range 0 to 16 that causes polygons that are physically coplanar to appear separate. Polygons with a high z-bias value will appear in front of polygons with a low value, without

requiring sorting for drawing order. Polygons with a value of 1 appear in front of polygons with a value of 0, and so on. The default value is zero.

D3DRENDERSTATE_RANGEFOGENABLE

TRUE to enable range-based fog. (The default value is FALSE, in which case the system uses depth-based fog.) In range-based fog, the distance of an object from the viewer is used to compute fog effects, not the depth of the object (that is, the z-coordinate) in the scene. In range-based fog, all fog methods work as usual, except that they use range instead of depth in the computations.

Range is the correct factor to use for fog computations, but depth is commonly used instead because range is expensive to compute and depth is generally already available. Using depth to calculate fog has the undesirable effect of having the 'fogginess' of peripheral objects change as the eye is rotated − in this case, the depth changes while the range remains constant.

This render state works only with **D3DVERTEX** vertices. When you specify **D3DLVERTEX** or **D3DTLVERTEX** vertices, the F (fog) component of the RGBF fog value should already be corrected for range.

Since no hardware currently supports per-pixel range-based fog, range correction is calculated at vertices.

D3DRENDERSTATE_ANISOTROPY

Integer value that enables a degree of anisotropic filtering. (This is used for bilinear or trilinear filtering.) The value determines the maximum aspect ratio of the sampling filter kernel. To determine the range of appropriate values, use the D3DPRASTERCAPS_ANISOTROPY flag in the **D3DPRIMCAPS** structure.

Anisotropy is the distortion visible in the texels of a 3-D object whose surface is oriented at an angle with respect to the plane of the screen. The anisotropy is measured as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

D3DRENDERSTATE_STIPPLEPATTERN00 through **D3DRENDERSTATE_STIPPLEPATTERN31**

Stipple pattern. Each render state applies to a separate line of the stipple pattern. Together, these render states specify a 32x32 stipple pattern.

D3DRENDERSTATE_FORCE_DWORD

Forces this enumerated type to be 32 bits in size.

See Also

D3DOPCODE, **D3DSTATE**

D3DSHADEMODE

The **D3DSHADEMODE** enumerated type describes the supported shade mode for the **D3DRENDERSTATE_SHADEMODE** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DSHADEMODE {
    D3DSHADE_FLAT         = 1,
    D3DSHADE_GOURAUD      = 2,
    D3DSHADE_PHONG        = 3,
    D3DSHADE_FORCE_DWORD  = 0x7fffffff,
} D3DSHADEMODE;
```

Members

D3DSHADE_FLAT
Flat shade mode. The color and specular component of the first vertex in the triangle are used to determine the color and specular component of the face. These colors remain constant across the triangle; that is, they aren't interpolated.

D3DSHADE_GOURAUD
Gouraud shade mode. The color and specular components of the face are determined by a linear interpolation between all three of the triangle's vertices.

D3DSHADE_PHONG
Phong shade mode is not currently supported.

D3DSHADE_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

D3DRENDERSTATETYPE

D3DTEXTUREADDRESS

The **D3DTEXTUREADDRESS** enumerated type describes the supported texture addressing modes for the **D3DRENDERSTATE_TEXTUREADDRESS** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DTEXTUREADDRESS {
    D3DTADDRESS_WRAP         = 1,
    D3DTADDRESS_MIRROR       = 2,
    D3DTADDRESS_CLAMP        = 3,
    D3DTADDRESS_BORDER       = 4,
    D3DTADDRESS_FORCE_DWORD  = 0x7fffffff,
} D3DTEXTUREADDRESS;
```

Members

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.

D3DTADDRESS_BORDER
Texture coordinates outside the range [0.0, 1.0] are set to the border color, which is a new render state corresponding to **D3DRENDERSTATE_BORDERCOLOR** in the **D3DRENDERSTATETYPE** enumerated type.

This member was introduced in DirectX 5.

D3DTADDRESS_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

For more information about using the **D3DRENDERSTATE_WRAPU** and **D3DRENDERSTATE_WRAPV** render states, see Textures.

See Also

D3DRENDERSTATETYPE

D3DTEXTUREBLEND

The **D3DTEXTUREBLEND** enumerated type defines the supported texture-blending modes. This enumerated type is used by the **D3DRENDERSTATE_TEXTUREMAPBLEND** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DTEXTUREBLEND {
    D3DTBLEND_DECAL         = 1,
    D3DTBLEND_MODULATE      = 2,
    D3DTBLEND_DECALALPHA    = 3,
    D3DTBLEND_MODULATEALPHA = 4,
    D3DTBLEND_DECALMASK     = 5,
    D3DTBLEND_MODULATEMASK  = 6,
    D3DTBLEND_COPY          = 7,
    D3DTBLEND_ADD           = 8,
    D3DTBLEND_FORCE_DWORD   = 0x7fffffff,
} D3DTEXTUREBLEND;
```

Members

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.

cPix = cTex
aPix = aTex

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; if the texture does not contain an alpha component, alpha values at the vertices in the source are interpolated between vertices.

*cPix = cSrc * cTex*
aPix = aTex

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:

$$C = (1 - A_t) C_o + A_t C_t$$

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.

*cPix = (cSrc * (10 - aTex)) + (aTex * cTex)*
aPix = aSrc

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.

*cPix = cSrc * cTex*
*aPix = aSrc * aTex*

D3DTBLEND_DECALMASK
Decal-mask texture-blending mode is supported.

```
cPix = lsb(aTex) ? cTex : cSrc
aPix = aSrc
```

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_MODULATEMASK
Modulate-mask texture-blending mode is supported.

```
cPix = lsb(aTex) ? cTex * cSrc : cSrc
aPix = aSrc
```

When the least-significant bit of the texture's alpha component is zero, the effect is as if texturing were disabled.

D3DTBLEND_COPY
Copy texture-blending mode is supported. This mode is an optimization for software rasterization; for applications using a HAL, it is equivalent to the **D3DTBLEND_DECAL** texture-blending mode.

To use copy mode, textures must use the same pixel format and palette format as the destination surface; otherwise nothing is rendered. Copy mode does no lighting and simply copies texture pixels to the screen. This is often a good technique for prelit textured scenes.

```
cPix = cTex
aPix = aTex
```

For more information, see Copy Texture-blending Mode.

D3DTBLEND_ADD
Add the Gouraud interpolants to the texture lookup with saturation semantics (that is, if the color value overflows it is set to the maximum possible value). This member was introduced in DirectX 5.

```
cPix = cTex + cSrc
aPix = aSrc
```

D3DTBLEND_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

In the formulas given for the members of this enumerated type, the placeholders have the following meanings:

- *cTex* is the color of the source texel
- *aTex* is the alpha component of the source texel
- *cSrc* is the interpolated color of the source primitive
- *aSrc* is the alpha component of the source primitive
- *cPix* is the new blended color value
- *aPix* is the new blended alpha value

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 preexisting 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

The **D3DTEXTUREFILTER** enumerated type defines the supported texture filter modes used by the **D3DRENDERSTATE_TEXTUREMAG** render state in the **D3DRENDERSTATETYPE** enumerated type.

```
typedef enum _D3DTEXTUREFILTER {
    D3DFILTER_NEAREST          = 1,
    D3DFILTER_LINEAR           = 2,
    D3DFILTER_MIPNEAREST       = 3,
    D3DFILTER_MIPLINEAR        = 4,
    D3DFILTER_LINEARMIPNEAREST = 5,
    D3DFILTER_LINEARMIPLINEAR  = 6,
    D3DFILTER_FORCE_DWORD   = 0x7fffffff,
} D3DTEXTUREFILTER;
```

Members

D3DFILTER_NEAREST
The texel with coordinates nearest to the desired pixel value is used. This is a point filter with no mipmapping.

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 2×2 area of texels surrounding the desired pixel is used. This is a bilinear filter with no mipmapping.

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
The closest mipmap level is chosen and a point filter is applied.

D3DFILTER_MIPLINEAR
The closest mipmap level is chosen and a bilinear filter is applied within it.

D3DFILTER_LINEARMIPNEAREST
The two closest mipmap levels are chosen and then a linear blend is used between point filtered samples of each level.

D3DFILTER_LINEARMIPLINEAR
The two closest mipmap levels are chosen and then combined using a bilinear filter.

D3DFILTER_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

Remarks

All of these filter modes are valid with the **D3DRENDERSTATE_TEXTUREMIN** render state, but only the first two (**D3DFILTER_NEAREST** and **D3DFILTER_LINEAR**) are valid with **D3DRENDERSTATE_TEXTUREMAG**.

D3DTRANSFORMSTATETYPE

The **D3DTRANSFORMSTATETYPE** enumerated type describes the transformation state for the **D3DOP_STATETRANSFORM** opcode in the **D3DOPCODE** enumerated type. This enumerated type is part of the **D3DSTATE** structure.

```
typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTRANSFORMSTATE_WORLD        = 1,
    D3DTRANSFORMSTATE_VIEW         = 2,
    D3DTRANSFORMSTATE_PROJECTION   = 3,
    D3DTRANSFORMSTATE_FORCE_DWORD  = 0x7fffffff,
} D3DTRANSFORMSTATETYPE;
```

Members

D3DTRANSFORMSTATE_WORLD,
D3DTRANSFORMSTATE_VIEW, and
D3DTRANSFORMSTATE_PROJECTION
Define the matrices for the world, view, and projection transformations. The default values are NULL (the identity matrices).

D3DTRANSFORMSTATE_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

D3DOPCODE, **D3DRENDERSTATETYPE**

D3DVERTEXTYPE

The **D3DVERTEXTYPE** enumerated type lists the vertex types that are supported by Direct3D.

```
typedef enum _D3DVERTEXTYPE {
    D3DVT_VERTEX       = 1,
    D3DVT_LVERTEX      = 2,
    D3DVT_TLVERTEX     = 3
    D3DVT_FORCE_DWORD  = 0x7fffffff,
};
```

Members

D3DVT_VERTEX
All the vertices in the array are of the **D3DVERTEX** type. This setting will cause transformation, lighting and clipping to be applied to the primitive as it is rendered.

D3DVT_LVERTEX
All the vertices in the array are of the **D3DLVERTEX** type. When used with this option, the primitive will have transformations applied during rendering.

D3DVT_TLVERTEX
All the vertices in the array are of the **D3DTLVERTEX** type. Rasterization only will be applied to this data.

D3DVT_FORCE_DWORD
Forces this enumerated type to be 32 bits in size.

See Also

IDirect3DDevice2::Begin, **IDirect3DDevice2::BeginIndexed**, **IDirect3DDevice2::DrawIndexedPrimitive**, **IDirect3DDevice2::DrawPrimitive**

Other Types

This section contains information about the following Direct3D Immediate Mode types that are neither structures nor enumerated types:

- **D3DCOLOR**
- **D3DVALUE**

D3DCOLOR

The **D3DCOLOR** type is the fundamental Direct3D color type.

```
typedef DWORD D3DCOLOR, D3DCOLOR, *LPD3DCOLOR;
```

See Also

D3DRGB, **D3DRGBA**

D3DVALUE

The **D3DVALUE** type is the fundamental Direct3D fractional data type.

```
typedef float D3DVALUE, *LPD3DVALUE;
```

Return Values

Errors are represented by negative values and cannot be combined. This table lists the values that can be returned by all Direct3D Immediate Mode methods. See the individual method descriptions for lists of the values each can return.

D3D_OK
D3DERR_BADMAJORVERSION
D3DERR_BADMINORVERSION
D3DERR_DEVICEAGGREGATED (new for DirectX 5)
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_INITFAILED (new for DirectX 5)
D3DERR_INBEGIN (new for DirectX 5)
D3DERR_INVALID_DEVICE (new for DirectX 5)
D3DERR_INVALIDCURRENTVIEWPORT (new for DirectX 5)
D3DERR_INVALIDPALETTE(new for DirectX 5)
D3DERR_INVALIDPRIMITIVETYPE (new for DirectX 5)
D3DERR_INVALIDRAMPTEXTURE (new for DirectX 5)
D3DERR_INVALIDVERTEXTYPE (new for DirectX 5)
D3DERR_LIGHT_SET_FAILED
D3DERR_LIGHTHASVIEWPORT (new for DirectX 5)
D3DERR_LIGHTNOTINTHISVIEWPORT (new for DirectX 5)
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_NOCURRENTVIEWPORT (new for DirectX 5)
D3DERR_NOTINBEGIN (new for DirectX 5)
D3DERR_NOVIEWPORTS (new for DirectX 5)
D3DERR_SCENE_BEGIN_FAILED
D3DERR_SCENE_END_FAILED
D3DERR_SCENE_IN_SCENE
D3DERR_SCENE_NOT_IN_SCENE
D3DERR_SETVIEWPORTDATA_FAILED

D3DERR_SURFACENOTINVIDMEM (new for DirectX 5)
D3DERR_TEXTURE_BADSIZE (new for DirectX 5)
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
D3DERR_VIEWPORTDATANOTSET (new for DirectX 5)
D3DERR_VIEWPORTHASNODEVICE (new for DirectX 5)
D3DERR_ZBUFF_NEEDS_SYSTEMMEMORY (new for DirectX 5)
D3DERR_ZBUFF_NEEDS_VIDEOMEMORY (new for DirectX 5)

About Retained Mode

This section describes Direct3D's Retained Mode, Microsoft's solution for real-time 3-D graphics on the personal computer. If you need to create a 3-D 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.

If you have written code that uses 3-D graphics before, many of the concepts underlying Retained Mode will be familiar to you. If, however, you are new to 3-D programming, you should pay close attention to Direct3D Retained-Mode Architecture, and you should read Getting Started. Whether you are new to 3-D 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 3-D 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.

Getting Started

The following sections describe some of the technical concepts you need to understand before you write programs that incorporate 3-D 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 Direct3D Retained-Mode Architecture.

If you are already experienced in producing 3-D graphics, simply scan the following sections for information that is unique to Direct3D Retained Mode.

3D Coordinate Systems

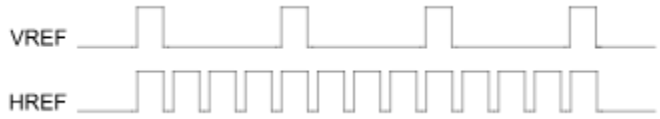
There are two varieties of Cartesian coordinate systems in 3-D graphics: left-handed and right-handed. In both coordinate systems, the positive x-axis points to the right and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your 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.

This section describes the Direct3D coordinate system and coordinate types that your application can use.

- Direct3D's Coordinate System
- U- and V-Coordinates

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

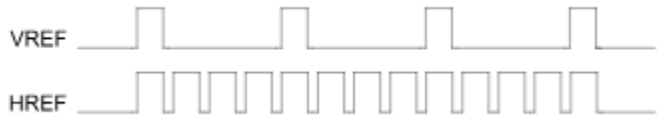
Direct3D also uses *texture coordinates*. These coordinates (u and v) are used when mapping textures onto an object. The v-vector describes the direction or orientation of the texture and lies along the z-axis. The u-vector (or the *up* vector) typically lies along the y-axis, with its origin at [0,0,0]. For more information about u- and v-coordinates, see IDirect3DRMWrap Interface.

3D Transformations

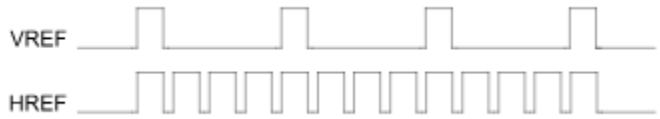
In programs that work with 3-D graphics, you can use geometrical transformations to:

- 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 4×4 matrix. In the following example, a matrix is used to reinterpret the point (x, y, z), producing the new point (x', y', z'):

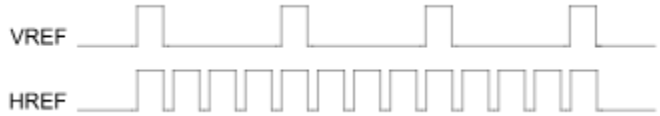


You perform the following operations on (x, y, z) and the matrix to produce the point (x', y', z'):



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:



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)
};
```

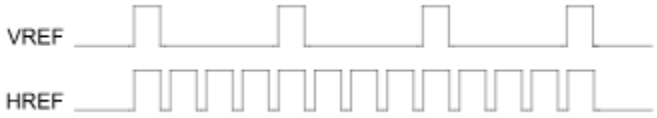
This section describes the 3-D transformations available to your applications through Direct3D.

- Translation
- Rotation
- Scaling

Other parts of this documentation also discuss transformations. You can find a general discussion of transformations in the section devoted to viewports in Retained Mode, Transformations. For a discussion of transformations in frames, see Transformations. Although each of these sections discusses Retained-Mode API, the architecture and mathematics of the transformations apply to both Retained Mode and Immediate Mode.

Translation

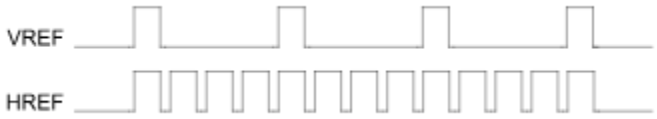
The following transformation translates the point (x, y, z) to a new point (x', y', z'):



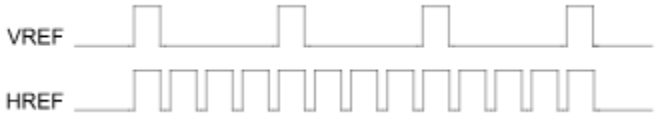
Rotation

The transformations described in this section are for left-handed coordinate systems, and so may be different from transformation matrices you have seen elsewhere.

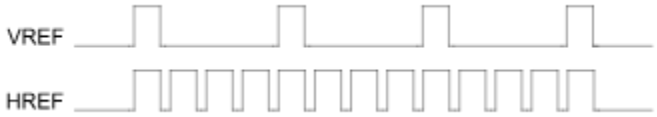
The following transformation rotates the point (x, y, z) around the x-axis, producing a new point (x', y', z'):



The following transformation rotates the point around the y-axis:



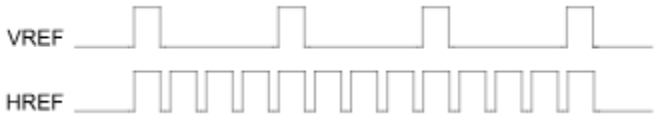
The following transformation rotates the point around the z-axis:



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'):



Polygons

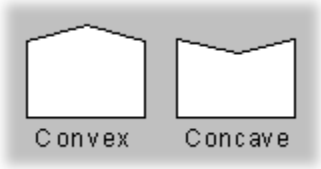
Three-dimensional objects in Direct3D are made up of meshes. A mesh is a set of faces, each of which is described by a simple polygon. The fundamental polygon type is the triangle. Although Retained-Mode applications can specify polygons with more than three vertices, the system translates these into triangles before the objects are rendered. Immediate-Mode applications must use triangles.

This section describes how your applications can use Direct3D polygons.

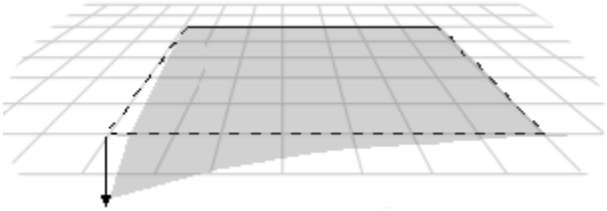
- Geometry Requirements
- Face and Vertex Normals
- Shade Modes
- Triangle Interpolants

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.

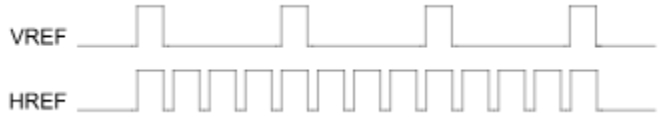


The three vertices of a triangle always describe a plane, but it is easy to accidentally create a nonplanar polygon by adding another vertex.

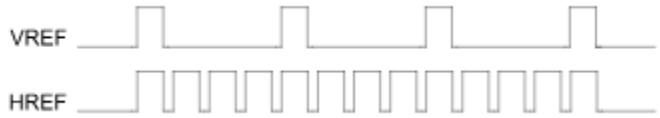


Face and Vertex Normals

Each face in a mesh has a perpendicular face *normal vector* 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.

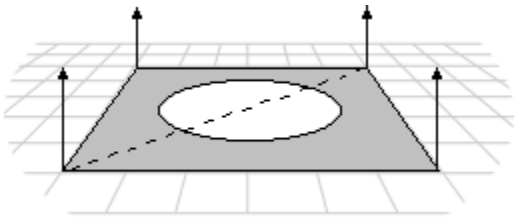


Shade Modes

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.

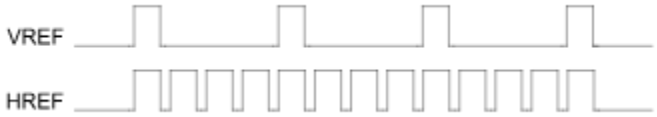
Note Phong shading is not currently supported.

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.

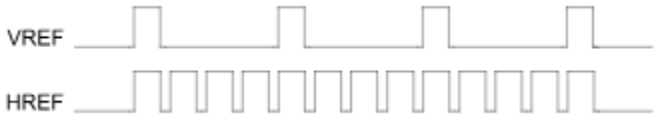


In this case, the Phong shade mode would calculate the value for each pixel and display the spotlight. The Gouraud shade mode, which interpolates between vertices, would miss the spotlight altogether; the face would be rendered as though the spotlight did not exist.

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 currently supported.

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.

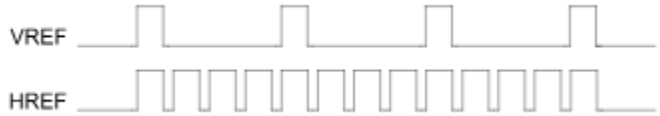
For example, if the red component of the color of vertex 1 were 0.8 and the red component of vertex 2 were 0.4, in the Gouraud shade mode and RGB color model the system would use interpolation to assign a red component of 0.6 to the pixel at the midpoint of the line between these vertices.

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.

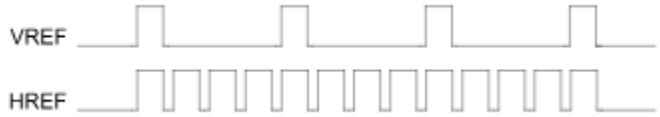
An application can use the **dwShadeCaps** member of the **D3DPRIMCAPS** structure to determine what forms of interpolation the current device driver supports.

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 v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the vertices of the second and fourth triangles 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 v1, v2, and v3 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v1, v4, and v5 to draw the third 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, Vertices, and Quaternions

Throughout Direct3D, vertices 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. Quaternions are an alternative to the matrix methods that are typically used for 3-D rotations. A quaternion represents an axis in 3-D space and a rotation around that axis. For example, a quaternion could represent a (1,1,2) axis and a rotation of 1 radian. Quaternions carry valuable information, but their true power comes from the two operations that you can perform on them: *composition* and *interpolation*.

Performing composition on quaternions is similar to combining them. The composition of two quaternions is notated as follows:

$$Q = q_1 \circ q_2$$

The composition of two quaternions applied to a geometry means "rotate the geometry around axis2 by rotation2, then rotate it around axis1 by rotation1." In this case, Q represents a rotation around a single axis that is the result of applying q2, then q1 to the geometry.
Using quaternion interpolation, an application can calculate a smooth and reasonable path from one axis and orientation to another. Therefore, interpolation between q1 and q2 provides a simple way to animate from one orientation to another.

When you use composition and interpolation together, they provide you with a simple way to manipulate a geometry in a manner that appears complex. For example, imagine that you have a geometry that you want to rotate to a given orientation. You know that you want to rotate it r2 degrees around axis2, then rotate it r1 degrees around axis1, but you don't know the final quaternion. By using composition, you could combine the two rotations on the geometry to get a single quaternion that is the result. Then, you could interpolate from the original to the composed quaternion to achieve a smooth transition from one to the other.

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. Additionally, the D3DRMQuaternionMultiply function composes quaternions and the D3DRMQuaternionSlerp performs spherical linear interpolation between two quaternions.

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.

Direct3D Retained-Mode Architecture

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 or Direct3DRMFrame2 object. 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.
Direct3DRMDevice2	This object represents the visual display destination for the renderer. Same as the Direct3DRMDevice object but with enhanced control of transparency.
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.
Direct3DRMFrame2	Extends the **Direct3DRMFrame** object by enabling access to the frame axes, bounding boxes, and materials. Also supports ray picking.
Direct3DRMInterpolator	This object stores actions and applies the actions to objects with automatic calculation of in-between values.
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.
Direct3DRMMeshBuilder2	This object allows you to work with individual vertices and faces in a mesh. Same as the **Direct3DRMMeshBuilder** object but with than the

	Direct3DRMMeshBuilder object.
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 2-D point.
Direct3DRMPicked2Array	This object identifies a visual object that corresponds to a given ray intersection.
Direct3DRMProgressiveMesh	This object consists of a coarse base mesh together with records describing how to incrementally refine the mesh. This allows a generalized level of detail to be set on the mesh as well as progressive download of the mesh from a remote source.
Direct3DRMShadow	This object defines a shadow.
Direct3DRMTexture	This object is a rectangular array of colored pixels.
Direct3DRMTexture2	This object is a rectangular array of colored pixels. Same as the **Direct3DRMTexture** object except that resources can be loaded from files other than the currently executing file, textures can be created from images in memory, and you can generate MIP maps.
Direct3DRMUserVisual	This object is defined by an application to provide functionality not otherwise available in the system.
Direct3DRMViewport	This object defines how the 3-D scene is rendered into a 2-D 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 IDirect3DRM 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
Direct3DRMDevice2	IDirect3DRMDevice2, IDirect3DRMWinDevice
Direct3DRMFace	IDirect3DRMFace
Direct3DRMFrame	IDirect3DRMFrame, **IDirect3DRMVisual**
Direct3DRMFrame2	IDirect3DRMFrame2, **IDirect3DRMVisual**
Direct3DRMInterpolator	IDirect3DRInterpolator
Direct3DRMLight	IDirect3DRMLight
Direct3DRMMaterial	IDirect3DRMMaterial
Direct3DRMMesh	IDirect3DRMMesh, **IDirect3DRMVisual**
Direct3DRMProgressiveMesh	IDirect3DRMProgressiveMesh, **IDirect3DRMVisual**
Direct3DRMMeshBuilder	IDirect3DRMMeshBuilder, **IDirect3DRMVisual**
Direct3DRMMeshBuilder2	IDirect3DRMMeshBuilder2, **IDirect3DRMVisual**
Direct3DRMShadow	IDirect3DRMShadow, **IDirect3DRMVisual**
Direct3DRMTexture	IDirect3DRMTexture, IDirect3DRMTexture2, **IDirect3DRMVisual**
Direct3DRMUserVisual	IDirect3DRMUserVisual, **IDirect3DRMVisual**
Direct3DRMViewport	IDirect3DRMViewport
Direct3DRMWrap	IDirect3DRMWrap

The following example 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. 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 or IDirect3DRMMeshBuilder2::GetFaces
IDirect3DRMFrameArray	IDirect3DRMPickedArray::GetPick,

	IDirect3DRMPicked2Array::GetPick, or IDirect3DRMFrame::GetChildren
IDirect3DRMLightArray	IDirect3DRMFrame::GetLights
IDirect3DRMObjectArray	IDirect3DRMInterpolator::GetAttachedObjects
IDirect3DRMPickedArray	IDirect3DRMViewport::Pick
IDirect3DRMPicked2Array	IDirect3DRMFrame2::RayPick
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 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.) Theoretically, an application could release a viewport without releasing the device (if it needed to add a new viewport to the device, for example), but whenever an application releases a device, it should release the viewport as well.

The reference count of a child or visual object increases whenever it is added to a 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.

Z Buffers in Retained-Mode

The order in the z buffer determines the order in which overlays clip each other. Overlays are assumed to be on top of all other screen components. Overlays that do not have a specified z-order behave in unpredictable ways when overlaying the same area on the primary surface. Direct3D Retained-Mode does not sort overlays if you do not have a z buffer. Overlays without a specified z-order are assumed to have a z-order of 0, and will appear in the order they are rendered. The possible z-order of overlays ranges from 0, which is just on top of the primary surface, to 4 billion, which is as close to the viewer as possible. An overlay with a z-order of 2 would obscure an overlay with a z-order of 1. No two overlays can have the same z-order.

IDirect3DRM and IDirect3DRM2 Interfaces

Applications use the methods of the **IDirect3DRM** interface to create Direct3DRM objects and work with system-level variables. For a reference to the methods of this interface, see IDirect3DRM or IDirect3DRM2.

Applications use the methods of the **IDirect3DRMDevice** and **IDirect3DRMDevice2** interfaces to interact with the output device. An **IDirect3DRMDevice** created from the **IDirect3DRM** interface works with an **IDirect3DDevice** Immediate-Mode device. An **IDirect3DRMDevice2** created from the **IDirect3DRM2** interface, or initialized by the IDirect3DRMDevice2::InitFromClipper, IDirect3DRMDevice::InitFromD3D, or IDirect3DRMDevice2::InitFromSurface method works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRM2 supports all the methods in **IDirect3DRM**. An additional method is included IDirect3DRM2::CreateProgressiveMesh. The IDirect3DRM2::CreateDeviceFromSurface methods, IDirect3DRM2::CreateDeviceFromD3D, and IDirect3DRM2::CreateDeviceFromClipper all create a DIRECT3DRMDEVICE2 object. The IDirect3DRM2::CreateViewport method creates a viewport on a DIRECT3DRMDEVICE2 object. The IDirect3DRM2::LoadTexture and IDirect3DRM2::LoadTextureFromResource methods load a DIRECT3DRMTEXTURE2 object.

The IDirect3DRM COM interface is created by calling the Direct3DRMCreate function. To access the IDirect3DRM2 COM interface, create an IDirect3DRM object with Direct3DRMCreate, then query for IDirect3DRM2 from IDirect3DRM.

The methods of the **IDirect3DRM** and **IDirect3DRM2** interfaces create the following objects:

- Animations and animation sets
- Devices
- Faces
- Frames
- Generic uninitialized objects
- Lights
- Materials
- Meshes and mesh builders
- Shadows
- Textures
- UserVisuals
- Viewports
- Wraps

In addition, the IDirect3DRM2::CreateProgressiveMesh creates a DIRECT3DRMPROGRESSIVEMESH object.

IDirect3DRMAnimation and IDirect3DRMAnimationSet Interfaces

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.

The IDirect3DRMAnimation::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.

IDirect3DRMDevice, IDirect3DRMDevice2, and IDirect3DRMDeviceArray Interfaces

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.

While an **IDirect3DRMDevice** interface, when created from the **IDirect3DRM** interface, works with an **IDirect3DDevice** Immediate-Mode device, an **IDirect3DRMDevice2** interface, when created from the **IDirect3DRM2** interface or initialized by the IDirect3DRMDevice2::InitFromClipper, IDirect3DRMDevice2::InitFromD3D2, or IDirect3DRMDevice2::InitFromSurface method, works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

The IDirect3DRMDevice2::InitFromClipper and IDirect3DRMDevice2::InitFromSurface methods use the IDirect3DRM2::CreateDevice method to create an IDirect3DRMDevice2 object. The IDirect3DRMDevice2::InitFromD3D2 method uses an **IDirect3D2** Immediate-Mode object and an **IDirect3DDevice2** Immediate-Mode device to initialize an **IDirect3DDevice2** Retained-Mode device.

You can still query back and forth between the IDirect3DRMDevice and IDirect3DRMDevice2 interfaces. The main difference is in how the underlying Immediate-Mode device is created.

The IDirect3DRMDevice2 interface contains all the methods found in the IDirect3DRMDevice interface, plus two additional ones that allow you to control transparency, IDirect3DRMDevice2::GetRenderMode and IDirect3DRMDevice2::SetRenderMode, and one additional initialization method IDirect3DRMDevice2::InitFromSurface.

For related information, see IDirect3DRMDevice and IDirect3DRMDevice2.

This section describes the options available to display Direct3D images to output devices.

- Quality
- Color Models
- Window Management

Quality

The device allows the scene and its component parts to be rendered with various degrees of realism. The device rendering quality is the maximum quality at which rendering can take place on the rendering surface of that device. Mesh, progressive mesh, and mesh builder objects can also have a specified rendering quality.

A device's or object's quality has three components: shade mode (flat or gouraud, phong is not yet implemented and will default to gouraud shading), lighting type (on or off), and fill mode (point, wireframe or solid).

You can set the quality of a device with IDirect3DRMDevice::SetQuality and IDirect3DRMDevice2::SetQuality methods. By default, the device quality is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMProgressiveMesh, Direct3DRMMeshBuilder, or Direct3DRMMeshBuilder2 object with their respective **SetQuality** methods, IDirect3DRMProgressiveMesh::SetQuality, IDirect3DRMMeshBuilder::SetQuality, and IDirect3DRMMeshBuilder2::SetQuality. By default, the quality of these objects is D3DRMRENDER_GOURAUD (gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

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.

Palettes are supported for textures, off-screen surfaces, and overlay surfaces, none of which is required to have the same palette as the primary surface. If a device supports a 4-bit indexed palette (16 colors) and you have 8-bit indexed art (256 colors), Retained Mode will render the art as 4-bit by taking the first 16 entries from your palette and remapping to those. Therefore, you should put your 16 preferred colors at the front of the palette if possible.

Window Management

For correct operation, applications must inform Direct3D when the 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.

IDirect3DRMFace and IDirect3DRMFaceArray Interfaces

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.

IDirect3DRMFrame, IDirect3DRMFrame2, and IDirect3DRMFrameArray Interfaces

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 method and specifying NULL for the first parameter.

The IDirect3DRMFrame2 interface is an extension of the IDirect3DRMFrame interface. IDirect3DRMFrame2 has methods that enable using materials, bounding boxes, and axes with frames. IDirect3DRMFrame2 also supports ray picking.

By using the IDirect3DRMFrame2::SetAxes method and using the right-handed projection types in the D3DRMPROJECTIONTYPE structure with the IDirect3DRMViewport::SetProjection method, you can enable right-handed projection.

For related information, see IDirect3DRMFrame and IDirect3DRMFrame2.

This section describes frames and how your application can use them.

- Hierarchies
- Transformations
- Motion
- Callback Functions

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 and is called a *root frame*.

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 think of the position and orientation of a frame relative to its parent frame as a linear transformation. This transformation takes vectors defined relative to the child frame and changes them to equivalent vectors defined relative to the parent.

Transformations can be represented by 4×4 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 model coordinates.

You can find a more general discussion of transformations in the section devoted to viewports, Transformations. For an overview of the mathematics of transformations, see 3D Transformations.

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.

IDirect3DRMInterpolator Interface

Interpolators provide a way of storing actions and applying them to objects with automatic calculation of in-between values. For example, you can set a scene's background color to red at time zero and green at time ten, and the interpolator will automatically tint successive scenes to blend from red to green. With an interpolator, you can blend colors, move objects smoothly between positions, morph meshes, and perform many other transformations.

In the Direct3D Retained-Mode implementation, interpolators are a generalization of the IDirect3DRMAnimation interface that increases the kinds of object parameters you can animate. While the IDirect3DRMAnimation interface allows animation of an object's position, size and orientation, the IDirect3DRMInterpolator interface further enables animation of colors, meshes, textures, and materials.

Interpolator Keys

The actions stored by the interpolator are called keys. A key is a stored procedure call and has an index associated with it. The interpolator automatically calculates between the key values.

Keys are stored in the interpolator by calling one of the supported interface methods that can be interpolated. The method and the parameter values passed to it make up the key. Methods Supported by the Interpolator supplies a list of supported methods.

Every key stored inside an interpolator has an index value. When the key is recorded, it is stamped with the current interpolator index value. The key's index value never changes once this value is set.

Interpolator Types

Objects can be attached to interpolators of an associated type; for example, a Mesh can be attached to a MeshInterpolator. The interpolator types are:

- FrameInterpolator
- LightInterpolator
- MaterialInterpolator
- MeshInterpolator
- TextureInterpolator
- ViewportInterpolator

Other interpolators can also be attached to an interpolator. When you change the index of an interpolator, it sets the indices of any attached interpolators to the same value.

Note that for MeshInterpolators, you add a **SetVertices** key to a MeshInterpolator object by calling **SetVertices** on the MeshInterpolator object's IDirect3DRMMesh interface. The group index used with **SetVertices** must correspond to a valid group index in the Mesh object or objects that the interpolator is applied to.

Interpolator Example

As an example, if you want to interpolate a frame's position, you will need a FrameInterpolator object with two interfaces, IDirect3DRMInterpolator and IDirect3DRMFrame.

```
pd3drm->CreateObject(CLSID_CDirect3DRMFrameInterpolator, 0,
IID_IDirect3DRMInterpolator, &pInterp);
pInterp->QueryInterface(IID_IDirect3DRMFrame, &pFrameInterp);
```

To add a position key to the interpolator, set the interpolator's internal index through the IDirect3DRMInterpolator interface, and record the position by calling the IDirect3DRMFrame::SetPosition method on the IDirect3DRMFrame interface. This method is applied to the interpolator rather than to a real frame. The function call and its parameters are stored in the interpolator as a new key with the current index.

```
pInterp->SetIndex(keytime);
pFrameInterp->SetPosition(NULL, keypos.x, keypos.y, keypos.z);
```

You can add more keys by repeating the sequence of setting the index with **SetIndex** followed by one or more object methods. To play actions back through a real frame, attach the frame to the interpolator.

```
pInterp->AttachObject(pRealFrame);
```

Now call **Interpolate** to set the position of the *pRealFrame* parameter using the interpolated position.

```
pInterp->Interpolate(time, NULL, D3DRMINTERPOLATIONSPLINE |
D3DRMINTERPOLATION_OPEN);
```

The interpolator will call the attached frame's **SetPosition** method, passing it a position it has calculated by interpolating (in this case, using a B-spline) between the nearest **SetPosition** keys.

Alternatively, you can use the immediate form of **Interpolate** and pass the object as the second parameter. This overrides any attached objects.

```
pInterp->Interpolate(time, pRealFrame, D3DRMINTERPOLATIONSPLINE |
D3DRMINTERPOLATION_OPEN);
```

You can use the same interpolator to store other keys such as orientation, scale, velocity, and color keys. Each property exists on a parallel timeline, and calling **Interpolate** assigns the interpolated value for each property to the attached frames.

It is possible to interpolate more than one method. For example, you can store **SetGroupColor** and **SetVertices** keys in the same interpolator. It is not possible to interpolate between keys of different methods, so they are stored in parallel execution threads called Key Chains. Also, if you specify two keys from different groups, such as **SetGroupColor**(0, black) and **SetGroupColor**(2, white), it does not make sense for the interpolator to generate an in-between action of **SetGroupColor**(1, gray) because the keys apply to different groups. In this case, the keys are also stored in separate chains.

Methods Supported by the Interpolator

Viewport

SetFront(value)
SetBack(value)
SetField(value)
SetPlane(left, right, bottom, top)

Frame and Frame2

SetPosition(reference*, x, y, z)
SetRotation(reference*, x, y, z, theta)
SetVelocity(reference*, x, y, z, withRotation*)
SetOrientation(reference*, dx, dy, dz, ux, uy, uz)
SetColor(color)
SetColorRGB(red, green, blue)
SetSceneBackground(color)
SetSceneBackgroundRGB(red, green, blue)
SetSceneFogColor(color)
SetSceneFogParams(start, end, density)
SetQuaternion(reference*, quat)

Mesh

Translate(x, y, z)
SetVertices(group*, index*, count*, vertices)
SetGroupColor(group*, color)
SetGroupColorRGB(group*, red, green, blue)

Light

SetColor(color)
SetColorRGB(red, green, blue)
SetRange(value)
SetUmbra(value)
SetPenumbra(value)
SetConstantAttenuation(value)
SetLinearAttenuation(value)
SetQuadraticAttenuation(value)

Texture and Texture2

SetDecalSize(width, height)
SetDecalOrigin(x, y)
SetDecalTransparentColor(color)

Material

SetPower(value)
SetSpecular(red, green, blue)
SetEmissive(red, green, blue)

*—Indicates keys with different values for this parameter are inserted in separate chains

An attempt to set a key of any unsupported method will result in a non-fatal D3DRMERR_BADOBJECT error.

Interpolator Index Span

The interpolator covers a span of index values. This index span is dictated by the following rules:

- The start of the span is the minimum of all key index values and the current index.
- The end of the span is the maximum of all key index values and the current index.

Interpolation Options

Interpolation can be performed with one or more of the following options:

D3DRMINTERPOLATION_CLOSED
D3DRMINTERPOLATION_LINEAR
D3DRMINTERPOLATION_NEAREST
D3DRMINTERPOLATION_OPEN
D3DRMINTERPOLATION_SLERPNORMALS
D3DRMINTERPOLATION_SPLINE
D3DRMINTERPOLATION_VERTEXCOLOR

If the interpolator is executed CLOSED, the interpolation is cyclic. The keys effectively repeat infinitely with a period equal to the index span. For compatibility with animations, any key with an index equal to the end of the span is ignored.

If the interpolation is OPEN, the first and last keys of each key chain fix the interpolated values outside of the index span.

The NEAREST, LINEAR, and SPLINE options specify how in-betweening is performed on each key chain. If NEAREST is specified the nearest key value is used. LINEAR performs linear interpolation between the 2 nearest keys. SPLINE uses a B-spline blending function on the 4 nearest keys.

The following two options affect only the interpolation of IDirect3DRMMesh::SetVertices:

- VERTEXCOLOR specifies that vertex colors should be interpolated.
- SLERPNORMALS specifies that vertex normals should be spherically interpolated (not currently implemented).

IDirect3DRMLight and IDirect3DRMLightArray Interfaces

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.

Retained Mode currently provides five types of light sources: ambient, directional, parallel point, point, and spotlight.

For a reference to the methods of the **IDirect3DRMLight** interface, see IDirect3DRMLight.

This section describes lighting effects available in Direct3D and how your application can use them.

- Ambient
- Directional
- Parallel Point
- Point
- 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

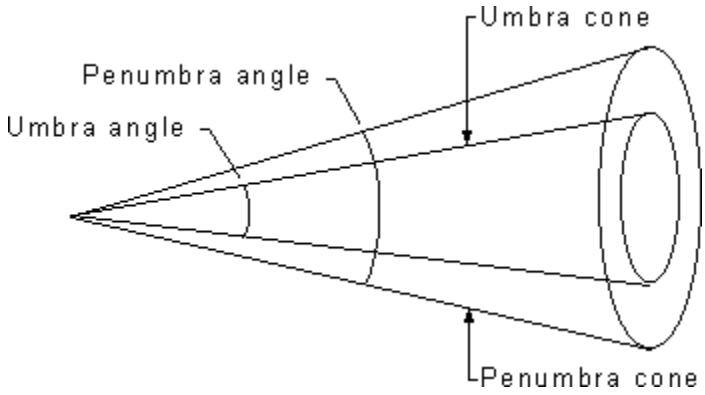
A *parallel point* light source illuminates objects with parallel light, but the orientation of the light is taken from the position of the parallel point light source. That is, like a directional light source, a parallel point light source has orientation, but it also has position. For example, two 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.



IDirect3DRMMaterial Interface

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.

For a reference to the methods of the **IDirect3DRMMaterial** interface, see IDirect3DRMMaterial.

IDirect3DRMMesh, IDirect3DRMMeshBuilder, and IDirect3DRMMeshBuilder2 Interfaces

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 2-D coordinates within a texture map.

You can manipulate meshes in Retained Mode by using three COM interfaces: IDirect3DRMMesh, IDirect3DRMMeshBuilder, and IDirect3DRMMeshBuilder2. **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.

IDirect3DRMMeshBuilder2 has all the functionality of **IDirect3DRMMeshBuilder** plus one enhanced and one added method. IDirect3DRMMeshBuilder2::GenerateNormals2 gives you more control over how normals are generated. IDirect3DRMMeshBuilder2::GetFace allows you to access a single face in a mesh.

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**, **IDirect3DRMMeshBuilder2**, 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 or the equivalent IDirect3DRMMeshBuilder2 methods. You can retrieve an individual face with the IDirect3DRMMeshBuilder2::GetFace.

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 yet available, 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 and IDirect3DRMObjectArray.

IDirect3DRMPickedArray and IDirect3DRMPicked2Array Interfaces

Picking is the process of searching for visuals in a scene, given a 2-D coordinate in a viewport or a vector in a frame.

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 topmost parent in the hierarchy first in the array.

You can use the IDirect3DRMFrame2::RayPick method to retrieve an IDirect3DRMPicked2Array interface, and then call the IDirect3DRMPicked2Array::GetPick method to retrieve an IDirect3DRMFrameArray interface, a visual object, and information about the object intersected by the ray, including the face and group identifiers, pick position, and horizontal and vertical texture coordinates for the vertex, vertex normal, and color of the object. The array of frames is the path through the hierarchy leading to the visual object.

IDirect3DRMProgressiveMesh Interface

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.

A progressive mesh is a mesh that is stored as a base mesh (a coarse version) and a set of records that are used to increasingly refine the mesh. This allows you to set the level of detail rendered for a mesh and also allows progressive download from remote sources.

Using the methods of the IDirect3DRMProgressiveMesh interface, you can set the number of vertices or faces to render and thereby control the render detail. You can also specify a minimum level of detail required for rendering. Normally, a progressive mesh is rendered once the base mesh is available, but with the IDirect3DRMProgressiveMesh::SetMinRenderDetail method you can specify that a greater level of detail is necessary before rendering. You can also build a **Direct3DRMMesh** object from a particular state of the progressive mesh using the IDirect3DRMProgressiveMesh::CreateMesh method.

You can load a progressive mesh from a file, resource, memory, or URL. Loading can be done synchronously or asynchronously. You can check the status of a download with the IDirect3DRMProgressiveMesh::GetLoadStatus method, and terminate a download with the IDirect3DRMProgressiveMesh::Abort method. If loading is asynchronous, it is up to the application to use events through the IDirect3DRMProgressiveMesh::RegisterEvents and IDirect3DRMProgressiveMesh::GetLoadStatus methods to find out how the load is progressing.

IDirect3DRMShadow Interface

Applications can produce an initialized and usable shadow simply by calling the IDirect3DRM::CreateShadow method. The IDirect3DRMShadow interface exists so that applications which create a shadow by using the IDirect3DRM::CreateObject method can initialize the shadow by calling the IDirect3DRMShadow::Init method.

IDirect3DRMTexture and IDirect3DRMTexture2 Interfaces

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.

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. To avoid unnecessary delays when creating textures, hold onto textures you want to use again, instead of creating them each time they're needed. For optimal performance, use a texture surface format that is supported by the device you are using. This will avoid a costly format conversion when the texture is created and any time it changes.

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 IDirect3DRMWrap Interface.

The **IDirect3DRMTexture2** interface is an extension of the IDirect3DRMTexture interface. The IDirect3DRMTexture2::InitFromResource2 method allows resources to be loaded from DLLs and executables other than the currently executing file. In addition, **IDirect3DRMTexture2** has two new methods. IDirect3DRMTexture2::InitFromImage creates a texture from an image in memory. This method is equivalent to IDirect3DRM::CreateTexture. IDirect3DRMTexture2::GenerateMIPMap generates a MIP map from a source image.

Textures are loaded from BMP and DIB (device-independent bitmap) files right-side up in IDirect3DRMTexture2::InitFromFile and IDirect3DRMTexture2::InitFromResource2, unlike IDirect3DRMTexture::InitFromFile and IDirect3DRMTexture::InitFromResource where they are loaded inverted.

For a reference to the methods of these interfaces, see IDirect3DRMTexture and IDirect3DRMTexture2.

This section describes the types of textures supported by Direct3D and how your application can use them.

- Decals
- Texture Colors
- Mipmaps
- Texture Filtering
- Texture Transparency
- Texture Format Selection Rules

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 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*.

Texture Format Selection Rules

When you use a device-independent source image to create a device-dependent texture surface for rendering, the rules are (in order of precedence):

1. Preserve RGB/palettized nature
2. Preserve alpha channel
3. Preserve bit depth or palette size
4. Preserve RBGA masks
5. Prefer 8-bit palettized or 16-bit RGB

For more information about texture pixel formats, see *Texture Map Formats*.

For related information, see IDirect3DRMTexture and IDirect3DRMTexture2.

IDirect3DRMUserVisual Interface

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 and the IDirect3DRMUserVisual::Init method to initialize the object.

IDirect3DRMViewport and IDirect3DRMViewportArray Interface

The viewport defines how the 3-D scene is rendered into a 2-D window. The viewport defines a rectangular area on a device that objects will be rendered into.

For a reference to the methods of this interface, see IDirect3DRMViewport.

This section describes the viewport, its components, and techniques for their use.

- Camera
- Viewing Frustum
- Transformations
- Picking

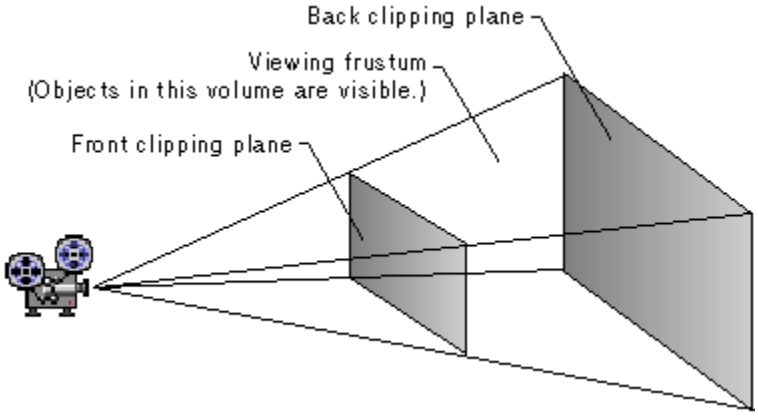
Camera

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 3-D volume in a scene positioned relative to the viewport's camera. For perspective viewing, the camera is positioned at the tip of an imaginary pyramid. This pyramid is intersected by two clipping planes, the front clipping plane and the back clipping plane. The volume in the pyramid between the front and back clipping planes is the viewing frustum. Only objects in the viewing frustum are visible.



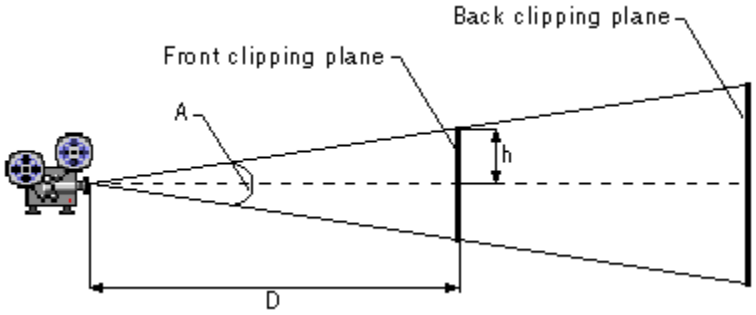
The z-axis of the camera runs from the tip of the pyramid to the center of the back clipping plane. Your application can set and retrieve the positions of the front and back clipping planes by using the IDirect3DRMViewport::SetFront, IDirect3DRMViewport::SetBack, IDirect3DRMViewport::GetFront, and IDirect3DRMViewport::GetBack methods.

The dimensions of the viewport on the front clipping plane determine the apparent focal length of the camera's lens. (You can also think of this as a way to set the magnification of objects in the frustum.) To set and retrieve proportional dimensions for the viewport on the front clipping plane, use the IDirect3DRMViewport::SetField and IDirect3DRMViewport::GetField methods. To set and retrieve arbitrary dimensions for the viewport on the front clipping plane, use the IDirect3DRMViewport::SetPlane and IDirect3DRMViewport::GetPlane methods.

You can use the following equation to determine the relationship between the height (or width) of the front clipping plane and the viewing angle:

$$A = 2\tan^{-1}\frac{h}{D}$$

In this formula, the viewing angle is *A*, the front clipping plane is a distance *D* from the camera, and the height or width of the front clipping plane is 2*h*. If the device is not square, and thus the clipping planes are not square, the viewing angle is calculated using half the height or half the width of the front clipping plane, whichever is larger. The scale factors are set to the major axis of the device so you don't get distorted objects. If this is not what you want, you need to set uniform scaling.

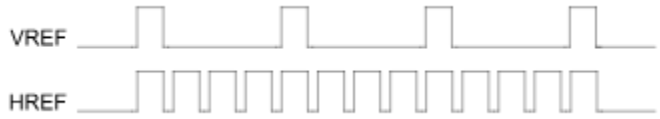


The viewing frustum is a pyramid only for perspective viewing. For orthographic viewing, the viewing frustum is cuboid. These viewing types (or projection types) are defined by the D3DRMPROJECTIONTYPE enumerated type and used by the IDirect3DRMViewport::GetProjection and IDirect3DRMViewport::SetProjection methods.

Transformations

To render objects with 3-D coordinates in a 2-D 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 values are the elements of the projection matrix. In these formulas, *h* is the half-height of the viewing frustum, *F* is the distance from the camera, and *D* is the position in z-coordinates of the front clipping plane:



After projection, the next step is clipping and the conversion of x and y to screen-pixel coordinates within a viewport. Use the D3DVIEWPORT data members for this. The viewport is a rectangular window on the rendering surface.

```
typedef struct _D3DVIEWPORT {
    DWORD    dwSize;
    DWORD    dwX;
    DWORD    dwY;
```

dwX and dwY specify the offset in screen pixels to the top left of the viewport on the surface.

```
    DWORD    dwWidth;
    DWORD    dwHeight;
```

dwWidth and dwHeight are the width and height of the viewport in screen pixels.

```
    D3DVALUE dvScaleX;
    D3DVALUE dvScaleY;
```

dvScaleX and dvScaleY are the scaling factors that are applied to the x and y values to yield screen coordinates. You would usually want to map the entire normalized perspective view volume onto the viewport using the following formulas:

```
  dvScaleX = dwWidth / 2
  dvScaleY = dwHeight / 2
```

X coordinates, for example, in the range of -1 to 1, will be scaled into the range of -dwWidth / 2 to dwWidth / 2. An offset of dwWidth / 2 is then added. This scaling occurs after clipping.

If the window is not square and you would like to preserve a correct aspect ratio, use the larger of the two window dimensions for both scaling values. You will also need to clip some of the view volume.

```
    D3DVALUE dvMaxX;
    D3DVALUE dvMaxY;
    D3DVALUE dvMinZ;
    D3DVALUE dvMaxZ;
```

These fields specify the clipping planes: x = dvMaxX, x = -dvMaxX, y = dvMaxY, y = -dvMaxY, z = dvMinZ, z = dvMaxZ. To display all of the view volume, for example, you would set dvMaxX = dvMaxY = dvMaxZ = 1 and dvMinZ = 0. As noted above, if you want to preserve the correct aspect ratio on a nonsquare window, you will need to clip some of the view volume. To do so, use the following equations. These equations also work with square viewports, so use them all the time.

```
dvMaxX = dwWidth / (2 * dvScaleX)
dvMaxY = dwHeight / (2 * dvScaleY)

} D3DVIEWPORT, *LPD3DVIEWPORT;
```

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.

For information about transformations for frames, see Transformations. For an overview of the mathematics of transformations, see 3D Transformations.

Picking

Picking is the process of searching for visuals in the scene given a 2-D 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.

IDirect3DRMVisual and IDirect3DRMVisualArray Interfaces

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.

IDirect3DRMWrap Interface

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.

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].

For a reference to the methods of the **IDirect3DRMWrap** interface, see IDirect3DRMWrap.

This section describes the wrapping flags and the four wrapping types:

- Wrapping Flags
- Flat
- Cylindrical
- Spherical
- Chrome

Wrapping Flags

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.

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:

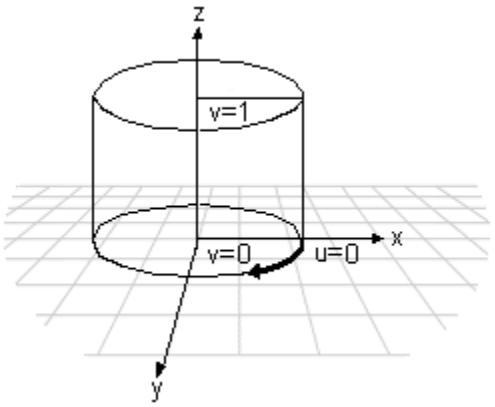
$$u = s_u x - o_u$$
$$v = s_v y - o_v$$

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:

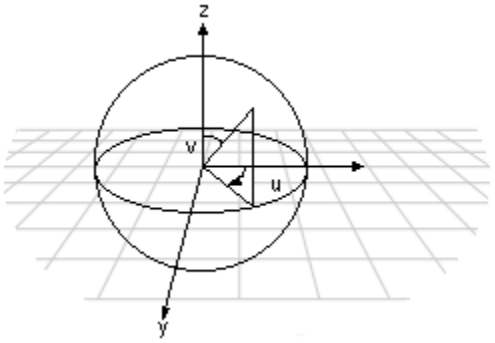
$$u = \frac{s_u}{2\pi} \tan^{-1} \frac{x}{y} - o_y$$

$$v = s_v z - o_v$$

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:

$$u = \frac{S_u}{2\pi} \tan^{-1} \frac{x}{y} - o_u$$

$$v = \frac{S_v}{\pi} \tan^{-1} \frac{z}{\sqrt{x^2 + y^2 + z^2}} - o_v$$

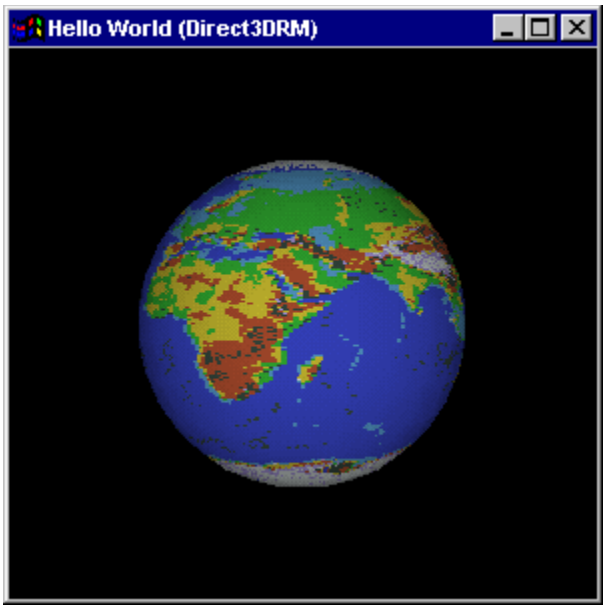
The scaling factors and texture origin will often not be needed here as the unscaled range of *u* and *v* is already 0 through 1.

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 is a tutorial that contains all the code for a simple Retained-Mode application. The following illustration is a single frame from the running animation:



The tutorial is divided into the following sections:

Making the Helworld Sample

This tutorial includes all the code for a working Direct3D Retained-Mode application. If you copy the code from this tutorial into a single .c file, you can compile and run it if:

- you have the Direct3D header files (d3d.h, d3drm.h, and so on) in your include path
- you link with the Winmm.lib, D3drm.lib, and Ddraw.lib static libraries
- your compiler can find the Sphere3.x file in the DirectX SDK's media directory
- you supply a bitmap called tutor.bmp

  Note that this bitmap must have pixel dimensions that are a power of 2; for example, a 16 by 16 pixel bitmap, a 128 by 1024 pixel bitmap, a 512 by 256 pixel bitmap, and so on.

Most of the code responsible for 3-D effects in this sample has been broken out into discrete functions so that you can modify parts of the system incrementally in your own experiments. You should look at the samples in the SDK for implementations of more complicated features of Direct3D.

The following topics describe some of the concerns that apply to the overall development of a simple Direct3D Retained-Mode application.

- Limitations of the Sample
- DirectDraw's Windowed Mode

Limitations of the Sample

This tutorial contains the Helworld.c example code, which creates a sphere, applies a texture to it, and rotates it in a window. This is the only C source file required to build this application. The only other files you need are Sphere3.x, a mesh file that ships in the Media directory of the DirectX SDK, and tutor.bmp, a bitmap you supply. (Note that you must have the Direct3D header files in your include path and link to the Winmm.lib, D3drm.lib, and Ddraw.lib static libraries.)

This tutorial is a simplied version fo the Globe sample that is part of the DirectX SDK. The Globe sample, like all the Direct3D Retained-Mode samples in the SDK, requires the inclusion of a file named Rmmain.cpp and a number of header files. In Helworld.c, the relevant parts of Rmmain.cpp have been converted to C from C++ and integrated into the source code.

The code shown in this tutorial should not be mistaken for production code. The only possible user interactions with the program are starting it, stopping it, and minimizing the window while it is running. Most of the error checking has been removed for the sake of clarity. The purpose of this example is analogous to the purpose of the beginning program that prints "Hello, world!" on the screen: to produce output with as little confusion as possible.

DirectDraw's Windowed Mode

Nearly all Direct3D applications will use DirectDraw to display their graphics on the screen. These applications will either use DirectDraw's full-screen (exclusive) mode or its windowed mode. The code in this documentation uses windowed mode. Although the full-screen mode offers some benefits in performance and convenience, it is much easier to debug code written in windowed mode. Most developers will write their code in windowed mode and port it to full-screen mode late in the development cycle, when most of the bugs have been worked out.

Definitions and Global Variables

Below are the first lines of the Helworld.c example. Helworld.c is the only C source file required to build this application.

Notice that INITGUID must be defined prior to all other includes and defines. This is a crucial point that is sometimes missed by developers who are new to DirectX.

```
//////////////////////////////////////////////////////////////////////
//
//  Copyright (C) 1996 Microsoft Corporation. All Rights Reserved.
//
//  File: Helworld.c
//
//  Simplified Direct3D Retained-Mode example, based on
//  the "Globe" SDK sample.
//
//////////////////////////////////////////////////////////////////////

#define INITGUID           // Must precede other defines and includes

#include <windows.h>
#include <malloc.h>        // Required by memset call
#include <d3drmwin.h>

#define MAX_DRIVERS 5           // Maximum D3D drivers expected

// Global variables

LPDIRECT3DRM lpD3DRM;           // Direct3DRM object
LPDIRECTDRAWCLIPPER lpDDClipper;// DirectDrawClipper object

struct _myglobs {
    LPDIRECT3DRMDEVICE dev;     // Direct3DRM device
    LPDIRECT3DRMVIEWPORT view;  // Direct3DRM viewport through which
                                // the scene is viewed
    LPDIRECT3DRMFRAME scene;    // Master frame in which others are
                                // placed
    LPDIRECT3DRMFRAME camera;   // Frame describing the user's POV

    GUID DriverGUID[MAX_DRIVERS];     // GUIDs of available D3D drivers
    char DriverName[MAX_DRIVERS][50]; // Names of available D3D drivers
    int  NumDrivers;                  // Number of available D3D drivers
    int  CurrDriver;                  // Number of D3D driver currently
                                      // being used

    BOOL bQuit;                 // Program is about to terminate
    BOOL bInitialized;          // All D3DRM objects are initialized
    BOOL bMinimized;            // Window is minimized

    int BPP;                    // Bit depth of the current display mode

} myglobs;

// Function prototypes.

static BOOL InitApp(HINSTANCE, int);
long FAR PASCAL WindowProc(HWND, UINT, WPARAM, LPARAM);
static BOOL EnumDrivers(HWND win);
static HRESULT WINAPI enumDeviceFunc(LPGUID lpGuid,
    LPSTR lpDeviceDescription, LPSTR lpDeviceName,
```

```
    LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc,
    LPVOID lpContext);
static DWORD BPPToDDBD(int bpp);
static BOOL CreateDevAndView(LPDIRECTDRAWCLIPPER lpDDClipper,
    int driver, int width, int height);
static BOOL SetRenderState(void);
static BOOL RenderLoop(void);
static BOOL MyScene(LPDIRECT3DRMDEVICE dev, LPDIRECT3DRMVIEWPORT view,
    LPDIRECT3DRMFRAME scene, LPDIRECT3DRMFRAME camera);
void MakeMyFrames(LPDIRECT3DRMFRAME lpScene, LPDIRECT3DRMFRAME lpCamera,
    LPDIRECT3DRMFRAME * lplpLightFrame1,
    LPDIRECT3DRMFRAME * lplpWorld_frame);
void MakeMyLights(LPDIRECT3DRMFRAME lpScene, LPDIRECT3DRMFRAME lpCamera,
    LPDIRECT3DRMFRAME lpLightFrame1,
    LPDIRECT3DRMLIGHT * lplpLight1, LPDIRECT3DRMLIGHT * lplpLight2);
void SetMyPositions(LPDIRECT3DRMFRAME lpScene,
    LPDIRECT3DRMFRAME lpCamera, LPDIRECT3DRMFRAME lpLightFrame1,
    LPDIRECT3DRMFRAME lpWorld_frame);
void MakeMyMesh(LPDIRECT3DRMMESHBUILDER * lplpSphere3_builder);
void MakeMyWrap(LPDIRECT3DRMMESHBUILDER sphere3_builder,
                LPDIRECT3DRMWRAP * lpWrap);
void AddMyTexture(LPDIRECT3DRMMESHBUILDER lpSphere3_builder,
                  LPDIRECT3DRMTEXTURE * lplpTex);
static void CleanUp(void);
```

Windows Setup and Initialization

This section describes the standard setup and initialization functions in a Windows program, as implemented in the Helworld.c sample code.

- The WinMain Function
- The InitApp Function
- The Main Window Procedure

The WinMain Function

The WinMain function in Helworld.c has only a few lines of code that are unique to an application that uses DirectDraw and Direct3D's Retained Mode. The InitApp and CleanUp functions are standard parts of a Windows program, although in the case of Helworld, they perform some unique tasks. The most important call in WinMain, from the perspective of Direct3D, is the call to the RenderLoop function. RenderLoop is responsible for drawing each new frame of the animation. For more information about the RenderLoop function, see The Rendering Loop.

```
//////////////////////////////////////////////////////////////////////
//
// WinMain
// Initializes the application and enters a message loop.
// The message loop renders the scene until a quit message is received.
//
//////////////////////////////////////////////////////////////////////

int PASCAL
WinMain (HINSTANCE this_inst, HINSTANCE prev_inst, LPSTR cmdline,
    int cmdshow)
{
    MSG     msg;
    HACCEL  accel = NULL;
    int     failcount = 0;  // Number of times RenderLoop has failed

    prev_inst;
    cmdline;

    // Create the window and initialize all objects needed to begin
    // rendering.

    if (!InitApp(this_inst, cmdshow))
        return 1;

    while (!myglobs.bQuit) {

        // Monitor the message queue until there are no pressing
        // messages.

        while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
            if (!TranslateAccelerator(msg.hwnd, accel, &msg)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }

        // If the app is not minimized, not about to quit, and D3DRM has
        // been initialized, begin rendering.

        if (!myglobs.bMinimized && !myglobs.bQuit &&
             myglobs.bInitialized) {

            // Attempt to render a frame. If rendering fails more than
            // twice, abort execution.

            if (!RenderLoop())
                ++failcount;
            if (failcount > 2) {
                CleanUp();
                break;
```

```
            }
        }
    }
    return msg.wParam;
}
```

The InitApp Function

The initialization function in Helworld.c creates the window class and main application window, just as in most Windows applications. After that, however, it does some work that is unique to applications using DirectDraw and Direct3D.

Now InitApp retrieves the current display's bits per pixel. This information is used to help set the rendering quality for the application. For more information, see Setting the Render State.

InitApp then calls the locally defined EnumDrivers function to determine what Direct3D drivers are available and to choose one. For more information about enumerating drivers, see Enumerating Device Drivers.

Next, the code calls the Direct3DRMCreate function to create an IDirect3DRM interface. It uses this interface in the calls to IDirect3DRM::CreateFrame and IDirect3DRMFrame::SetPosition that create the scene and camera frames, and position the camera in the scene.

A DirectDrawClipper object makes it simple to manage the clipping planes that control which parts of a 3-D scene are visible. Helworld.c calls the **DirectDrawCreateClipper** function to create an interface, and then uses the **IDirectDrawClipper::SetHWnd** method to set the window handle that obtains the clipping information.

Now the InitApp function calls the locally defined CreateDevAndView function to create the Direct3D device and viewport. For more information about this function, see Creating the Device and Viewport.

When the entire supporting structure of a Direct3D application has been put into place, the details of the 3-D scene can be constructed. The MyScene function does this work. For more information about MyScene, see Creating the Device and Viewport.

Finally, just as in a standard initialization function, InitApp shows and updates the window.

```
//////////////////////////////////////////////////////////////////////
//
// InitApp
// Creates window and initializes all objects necessary to begin
// rendering.
//
//////////////////////////////////////////////////////////////////////

static BOOL
InitApp(HINSTANCE this_inst, int cmdshow)
{
    HWND win;
    HDC hdc;
    WNDCLASS wc;
    RECT rc;

    // Set up and register the window class.

    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = sizeof(DWORD);
    wc.hInstance = this_inst;
    wc.hIcon = LoadIcon(this_inst, "AppIcon");
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "D3DRM Example";
    if (!RegisterClass(&wc))
        return FALSE;

    // Initialize the global variables.
```

```
memset(&myglobs, 0, sizeof(myglobs));

// Create the window.

win =
    CreateWindow
    (   "D3DRM Example",            // Class
        "Hello World (Direct3DRM)", // Title bar
        WS_VISIBLE | WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
             WS_MINIMIZEBOX | WS_MAXIMIZEBOX,
        CW_USEDEFAULT,              // Init. x pos
        CW_USEDEFAULT,              // Init. y pos
        300,                        // Init. x size
        300,                        // Init. y size
        NULL,                       // Parent window
        NULL,                       // Menu handle
        this_inst,                  // Program handle
        NULL                        // Create parms
    );
if (!win)
    return FALSE;

// Record the current display bits-per-pixel.

hdc = GetDC(win);
myglobs.BPP = GetDeviceCaps(hdc, BITSPIXEL);
ReleaseDC(win, hdc);

// Enumerate the D3D drivers and select one.

if (!EnumDrivers(win))
    return FALSE;

// Create the D3DRM object and the D3DRM window object.

lpD3DRM = NULL;
Direct3DRMCreate(&lpD3DRM);

// Create the master scene frame and camera frame.

lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, NULL, &myglobs.scene);
lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, myglobs.scene,
    &myglobs.camera);
myglobs.camera->lpVtbl->SetPosition(myglobs.camera, myglobs.scene,
    D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0));

// Create a DirectDrawClipper object and associate the
// window with it.

DirectDrawCreateClipper(0, &lpDDClipper, NULL);
lpDDClipper->lpVtbl->SetHWnd(lpDDClipper, 0, win);

// Create the D3DRM device by using the selected D3D driver.

GetClientRect(win, &rc);
if (!CreateDevAndView(lpDDClipper, myglobs.CurrDriver, rc.right,
        rc.bottom)) {
    return FALSE;
}
```

```
    // Create the scene to be rendered.

    if (!MyScene(myglobs.dev, myglobs.view, myglobs.scene,
            myglobs.camera))
        return FALSE;

    myglobs.bInitialized = TRUE;  // Initialization completed

    // Display the window.

    ShowWindow(win, cmdshow);
    UpdateWindow(win);

    return TRUE;
}
```

The Main Window Procedure

The Helworld.c sample has a very simple main window procedure—after all, the application allows practically no user interaction.

When the window procedure receives a WM_DESTROY message, it calls the CleanUp function, just as you would expect.

When it receives a WM_ACTIVATE message, it retrieves an IDirect3DRMWinDevice interface and calls the IDirect3DRMWinDevice::HandleActivate method to ensure that the colors are correct in the active rendering window. Similarly, the function reacts to a WM_PAINT message by calling the IDirect3DRMWinDevice::HandlePaint method.

```
//////////////////////////////////////////////////////////////////////
//
// WindowProc
// Main window message handler.
//
//////////////////////////////////////////////////////////////////////

LONG FAR PASCAL WindowProc(HWND win, UINT msg,
    WPARAM wparam, LPARAM lparam)
{
    RECT r;
    PAINTSTRUCT ps;
    LPDIRECT3DRMWINDEVICE lpD3DRMWinDev;

    switch (msg)    {

    case WM_DESTROY:
        CleanUp();
        break;

    case WM_ACTIVATE:
        {

        // Create a Windows-specific D3DRM window device to handle this
        // message.

        LPDIRECT3DRMWINDEVICE lpD3DRMWinDev;
        if (!myglobs.dev)
            break;
        myglobs.dev->lpVtbl->QueryInterface(myglobs.dev,
            &IID_IDirect3DRMWinDevice, (void **) &lpD3DRMWinDev);
        lpD3DRMWinDev->lpVtbl->HandleActivate(lpD3DRMWinDev,
            (WORD) wparam);
        lpD3DRMWinDev->lpVtbl->Release(lpD3DRMWinDev);
        }
        break;

    case WM_PAINT:
        if (!myglobs.bInitialized || !myglobs.dev)
            return DefWindowProc(win, msg, wparam, lparam);

        // Create a Windows-specific D3DRM window device to handle this
        // message.

        if (GetUpdateRect(win, &r, FALSE)) {
            BeginPaint(win, &ps);
            myglobs.dev->lpVtbl->QueryInterface(myglobs.dev,
                &IID_IDirect3DRMWinDevice, (void **) &lpD3DRMWinDev);
```

```
            if (FAILED(lpD3DRMWinDev->lpVtbl->HandlePaint(lpD3DRMWinDev,
                ps.hdc)))
            lpD3DRMWinDev->lpVtbl->Release(lpD3DRMWinDev);
            EndPaint(win, &ps);
        }
        break;
    default:
        return DefWindowProc(win, msg, wparam, lparam);
    }
    return 0L;
}
```

Enumerating Device Drivers

Applications that use Direct3D always enumerate the available drivers and choose the one that best matches their needs. The following sections each describe one of the functions that perform this task:

- The EnumDrivers Function
- The enumDeviceFunc Callback Function
- The BPPToDDBD Helper Function

The EnumDrivers Function

The EnumDrivers function is called by the InitApp function just before InitApp creates the application's scene and camera.

The COM interface is really an interface to a DirectDraw object, so the first thing this enumeration function does is call the **DirectDrawCreate** function to create a DirectDrawObject. Then EnumDrivers uses the **QueryInterface** method to create an **IDirect3D** interface. Notice that the C implementation of **QueryInterface** requires that you pass the address of the interface identifier as the second parameter, not simply the constant itself (as in the C++ implementation).

The enumeration is handled by the **IDirect3D::EnumDevices** method, which depends on the locally defined enumDeviceFunc callback function. For more information about this callback function, see The enumDeviceFunc Callback Function.

Notice that **IDirect3D::EnumDevices** is a Direct3D method, not a Direct3DRM method; there is no enumeration method in the Retained-Mode API. This is a good example of the natural use of both Retained-Mode and Immediate-Mode methods in a single application.

```
//////////////////////////////////////////////////////////////////////
//
// EnumDrivers
// Enumerate the available D3D drivers and choose one.
//
//////////////////////////////////////////////////////////////////////

static BOOL
EnumDrivers(HWND win)
{
    LPDIRECTDRAW lpDD;
    LPDIRECT3D lpD3D;
    HRESULT rval;

    // Create a DirectDraw object and query for the Direct3D interface
    // to use to enumerate the drivers.

    DirectDrawCreate(NULL, &lpDD, NULL);
    rval = lpDD->lpVtbl->QueryInterface(lpDD, &IID_IDirect3D,
        (void**) &lpD3D);
    if (rval != DD_OK) {
        lpDD->lpVtbl->Release(lpDD);
        return FALSE;
    }

    // Enumerate the drivers, setting CurrDriver to -1 to initialize the
    // driver selection code in enumDeviceFunc.

    myglobs.CurrDriver = -1;
    lpD3D->lpVtbl->EnumDevices(lpD3D, enumDeviceFunc,
        &myglobs.CurrDriver);

    // Ensure at least one valid driver was found.

    if (myglobs.NumDrivers == 0) {
        return FALSE;
    }
    lpD3D->lpVtbl->Release(lpD3D);
    lpDD->lpVtbl->Release(lpDD);

    return TRUE;
}
```

The enumDeviceFunc Callback Function

The enumDeviceFunc callback function is of the **D3DENUMDEVICESCALLBACK** type, as defined in the D3dcaps.h header file. The system calls this function with identifiers and names for each Direct3D driver in the system, as well as the hardware and emulated capabilities of the driver.

The callback function uses the **dcmColorModel** member of the **D3DDEVICEDESC** structure to determine whether to examine the hardware or emulated driver description; if the member has been filled by the hardware description, the function consults the hardware description.

Next, the callback function determines whether the driver being enumerated can render in the current bit depth. If not, the function returns D3DENUMRET_OK to skip the rest of the process for this driver and continue the enumeration with the next driver. The callback function uses the locally defined BPPToDDBD function to compare the reported bit depth against the bits-per-pixel retrieved by the call to the **GetDeviceCaps** function in the InitApp function. (BPPToDDBD stands for bits-per-pixel to DirectDraw bit-depth.) For the code for this function, see The BPPToDDBD Helper Function.

If the driver being enumerated passes a few simple tests, other parts of **D3DDEVICEDESC** are examined. The callback function will choose hardware over software emulation, and RGB lighting capabilities over monochromatic lighting capabilities.

```
//////////////////////////////////////////////////////////////////////
//
// enumDeviceFunc
// Callback function that records each usable D3D driver's name
// and GUID. Chooses a driver and sets *lpContext to this driver.
//
//////////////////////////////////////////////////////////////////////

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 should be consulted.

    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 name and GUID.

    memcpy(&myglobs.DriverGUID[myglobs.NumDrivers], lpGuid,
        sizeof(GUID));
    lstrcpy(&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 the 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 the start driver are the same type, and
            // this driver is mono whereas the 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);
}
```

The BPPToDDBD Helper Function

The enumDeviceFunc callback function uses the BPPToDDBD helper function to convert the stored bits-per-pixel the current device supports to a form that can be compared against the bit depth for the driver being enumerated. For more information about enumDeviceFunc, see The enumDeviceFunc Callback Function.

```
//////////////////////////////////////////////////////////////////////
//
// BPPToDDBD
// Converts bits-per-pixel to a DirectDraw bit-depth flag.
//
//////////////////////////////////////////////////////////////////////

static DWORD
BPPToDDBD(int bpp)
{
    switch(bpp) {
        case 1:
            return DDBD_1;
        case 2:
            return DDBD_2;
        case 4:
            return DDBD_4;
        case 8:
            return DDBD_8;
        case 16:
            return DDBD_16;
        case 24:
            return DDBD_24;
        case 32:
            return DDBD_32;
        default:
            return 0;
    }
}
```

Setting up the 3-D Environment

This section describes the code in Helworld.c that establishes the 3-D environment. The following sections describe the two functions that perform this task:

- Creating the Device and Viewport
- Setting the Render State

These functions do not populate the 3-D environment with objects, frames, and lights. That is done by the MyScene function and the functions it calls. For information about filling up the 3-D environment, see Creating the Scene.

Creating the Device and Viewport

The Direct3D device and viewport are created as part of the application's initialization. After creating a DirectDrawClipper object, the InitApp function calls CreateDevAndView, passing as parameters the DirectDrawClipper object, the driver that was chosen, and the dimensions of the client rectangle.

The CreateDevAndView function uses the IDirect3DRM::CreateDeviceFromClipper method to create a Direct3DRM device, using the driver that was selected by the enumeration process. It uses this IDirect3DRMDevice interface to retrieve the device's width and height, by calling IDirect3DRMDevice::GetWidth and IDirect3DRMDevice::GetHeight methods. After it has retrieved this information, it calls the IDirect3DRM::CreateViewport method to retrieve the IDirect3DRMViewport interface.

When CreateDevAndView has called the IDirect3DRMViewport::SetBack method to set the back clipping plane of the viewport, it calls the locally defined SetRenderState function. SetRenderState is described in the next section, Setting the Render State.

```
////////////////////////////////////////////////////////////////
//
// CreateDevAndView
// Create the D3DRM device and viewport with the given D3D driver and
// with the specified size.
//
////////////////////////////////////////////////////////////////

static BOOL
CreateDevAndView(LPDIRECTDRAWCLIPPER lpDDClipper, int driver,
    int width, int height)
{
    HRESULT rval;

    // Create the D3DRM device from this window by using the specified
    // D3D driver.

    lpD3DRM->lpVtbl->CreateDeviceFromClipper(lpD3DRM, lpDDClipper,
        &myglobs.DriverGUID[driver], width, height, &myglobs.dev);

    // Create the D3DRM viewport by using the camera frame.  Set the
    // background depth to a large number. The width and height
    // might have been slightly adjusted, so get them from the device.

    width = myglobs.dev->lpVtbl->GetWidth(myglobs.dev);
    height = myglobs.dev->lpVtbl->GetHeight(myglobs.dev);
    rval = lpD3DRM->lpVtbl->CreateViewport(lpD3DRM, myglobs.dev,
        myglobs.camera, 0, 0, width, height, &myglobs.view);
    if (rval != D3DRM_OK) {
        myglobs.dev->lpVtbl->Release(myglobs.dev);
        return FALSE;
    }
    rval = myglobs.view->lpVtbl->SetBack(myglobs.view, D3DVAL(5000.0));
    if (rval != D3DRM_OK) {
        myglobs.dev->lpVtbl->Release(myglobs.dev);
        myglobs.view->lpVtbl->Release(myglobs.view);
        return FALSE;
    }

    // Set the render quality, fill mode, lighting state,
    // and color shade info.

    if (!SetRenderState())
        return FALSE;
```

```
    return TRUE;
}
```

Setting the Render State

Direct3D is a *state machine*; applications set up the state of the lighting, rendering, and transformation modules and then pass data through them. This architecture is integral to Immediate Mode, but it is partially hidden by the Retained-Mode API. The SetRenderState function is a simple way to set the rendering state for a Retained-Mode application.

First, SetRenderState calls the IDirect3DRMDevice::SetQuality method, specifying that the lights are on, that the fill mode is solid, and that Gouraud shading should be used. At this point, applications that need to change the dithering mode or texture quality can call the IDirect3DRMDevice::SetDither or IDirect3DRMDevice::SetTextureQuality methods.

```
////////////////////////////////////////////////////////////////////
//
// SetRenderState
// Set the render quality and shade information.
//
////////////////////////////////////////////////////////////////////

BOOL
SetRenderState(void)
{
    HRESULT rval;

    // Set the render quality (light toggle, fill mode, shade mode).

    rval = myglobs.dev->lpVtbl->SetQuality(myglobs.dev,
        D3DRMLIGHT_ON | D3DRMFILL_SOLID | D3DRMSHADE_GOURAUD);
    if (rval != D3DRM_OK) {
        return FALSE;
    }

    // If you want to change the dithering mode, call SetDither here.

    // If you want a texture quality other than D3DRMTEXTURE_NEAREST
    // (the default value), call SetTextureQuality here.

   return TRUE;
}
```

The Rendering Loop

The WinMain function calls the RenderLoop function to draw each new frame of the animation. The RenderLoop function performs a few simple tasks:

- Calls the IDirect3DRMFrame::Move method to apply the rotations and velocities for all frames in the hierarchy.
- Calls the IDirect3DRMViewport::Clear method to clear the current viewport, setting it to the current background color.
- Calls the IDirect3DRMViewport::Render method to render the current scene into the current viewport.
- Calls the IDirect3DRMDevice::Update method to copy the rendered image to the display.

```
////////////////////////////////////////////////////////////////////
//
// RenderLoop
// Clear the viewport, render the next frame, and update the window.
//
////////////////////////////////////////////////////////////////////

static BOOL
RenderLoop()
{
    HRESULT rval;

    // Tick the scene.

    rval = myglobs.scene->lpVtbl->Move(myglobs.scene, D3DVAL(1.0));
    if (rval != D3DRM_OK) {
        return FALSE;
    }

    // Clear the viewport.

    rval = myglobs.view->lpVtbl->Clear(myglobs.view);
    if (rval != D3DRM_OK) {
        return FALSE;
    }

    // Render the scene to the viewport.

    rval = myglobs.view->lpVtbl->Render(myglobs.view, myglobs.scene);
    if (rval != D3DRM_OK) {
        return FALSE;
    }

    // Update the window.

    rval = myglobs.dev->lpVtbl->Update(myglobs.dev);
    if (rval != D3DRM_OK) {
        return FALSE;
    }
    return TRUE;
}
```

Creating the Scene

After setting up the 3-D environment—choosing a device driver, creating the 3-D device and viewport, setting the rendering state, and so on—Helworld.c calls a series of functions to populate this 3-D environment with objects, frames, and lights:

The MyScene Function

The MyScene function in Helworld.c corresponds to the BuildScene function that is implemented in all the Direct3D samples in the DirectX SDK. This is where all the work occurs that displays unique objects with unique textures and lighting effects.

The MyScene function calls a series of locally defined functions that set up the separate features of the scene that is being created. These functions are:

- MakeMyFrames
- MakeMyLights
- SetMyPositions
- MakeMyMesh
- MakeMyWrap
- AddMyTexture

When these functions have set up the visual object, MyScene calls the IDirect3DRMFrame::AddVisual method to add the object to the environment's world frame. After adding the visual object, MyScene no longer needs the interfaces it has created, so it calls the **Release** method repeatedly to release them all.

```
////////////////////////////////////////////////////////////////////
//
// MyScene
// Calls the functions that create the frames, lights, mesh, and
// texture. Releases all interfaces on completion.
//
////////////////////////////////////////////////////////////////////

BOOL
MyScene(LPDIRECT3DRMDEVICE dev, LPDIRECT3DRMVIEWPORT view,
    LPDIRECT3DRMFRAME lpScene, LPDIRECT3DRMFRAME lpCamera)
{
    LPDIRECT3DRMFRAME lpLightframe1 = NULL;
    LPDIRECT3DRMFRAME lpWorld_frame = NULL;
    LPDIRECT3DRMLIGHT lpLight1      = NULL;
    LPDIRECT3DRMLIGHT lpLight2      = NULL;
    LPDIRECT3DRMTEXTURE lpTex       = NULL;
    LPDIRECT3DRMWRAP lpWrap         = NULL;
    LPDIRECT3DRMMESHBUILDER lpSphere3_builder = NULL;

    MakeMyFrames(lpScene, lpCamera, &lpLightframe1, &lpWorld_frame);
    MakeMyLights(lpScene, lpCamera, lpLightframe1, &lpLight1,
        &lpLight2);
    SetMyPositions(lpScene, lpCamera, lpLightframe1, lpWorld_frame);
    MakeMyMesh(&lpSphere3_builder);
    MakeMyWrap(lpSphere3_builder, &lpWrap);
    AddMyTexture(lpSphere3_builder, &lpTex);

    // If you need to create a material (for example, to create
    // a shiny surface), call CreateMaterial and SetMaterial here.

    // Now that the visual object has been created, add it
    // to the world frame.

    lpWorld_frame->lpVtbl->AddVisual(lpWorld_frame,
        (LPDIRECT3DRMVISUAL) lpSphere3_builder);

    lpLightframe1->lpVtbl->Release(lpLightframe1);
    lpWorld_frame->lpVtbl->Release(lpWorld_frame);
```

```
    lpSphere3_builder->lpVtbl->Release(lpSphere3_builder);
    lpLight1->lpVtbl->Release(lpLight1);
    lpLight2->lpVtbl->Release(lpLight2);
    lpTex->lpVtbl->Release(lpTex);
    lpWrap->lpVtbl->Release(lpWrap);

    return TRUE;
}
```

The MakeMyFrames Function

The MyScene function calls the MakeMyFrames function to create the frames for the directional light and the world frame used in Helworld.c. MakeMyFrames does this work by calling the IDirect3DRM::CreateFrame method.

```
//////////////////////////////////////////////////////////////////////
//
// MakeMyFrames
// Create frames used in the scene.
//
//////////////////////////////////////////////////////////////////////

void MakeMyFrames(LPDIRECT3DRMFRAME lpScene, LPDIRECT3DRMFRAME lpCamera,
    LPDIRECT3DRMFRAME * lplpLightFrame1,
    LPDIRECT3DRMFRAME * lplpWorld_frame)
{
    lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, lpScene, lplpLightFrame1);
    lpD3DRM->lpVtbl->CreateFrame(lpD3DRM, lpScene, lplpWorld_frame);
}
```

The MakeMyLights Function

The MyScene function calls the MakeMyLights function to create the directional and ambient lights used in Helworld.c. MakeMyLights calls the IDirect3DRM::CreateLightRGB and IDirect3DRMFrame::AddLight methods to create a bright directional light and add it to a light frame, and to create a dim ambient light and add it to the entire scene. (Ambient lights are always associated with an entire scene.)

```
////////////////////////////////////////////////////////////////////
//
// MakeMyLights
// Create lights used in the scene.
//
////////////////////////////////////////////////////////////////////

void MakeMyLights(LPDIRECT3DRMFRAME lpScene, LPDIRECT3DRMFRAME lpCamera,
    LPDIRECT3DRMFRAME lpLightFrame1,
    LPDIRECT3DRMLIGHT * lplpLight1, LPDIRECT3DRMLIGHT * lplpLight2)
{
    lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM, D3DRMLIGHT_DIRECTIONAL,
        D3DVAL(0.9), D3DVAL(0.9), D3DVAL(0.9), lplpLight1);

    lpLightFrame1->lpVtbl->AddLight(lpLightFrame1, *lplpLight1);

    lpD3DRM->lpVtbl->CreateLightRGB(lpD3DRM, D3DRMLIGHT_AMBIENT,
        D3DVAL(0.1), D3DVAL(0.1), D3DVAL(0.1), lplpLight2);

    lpScene->lpVtbl->AddLight(lpScene, *lplpLight2);
}
```

The SetMyPositions Function

The MyScene function calls the SetMyPositions function to set the positions and orientations of the frames used in Helworld.c. SetMyPositions does this work by calling the IDirect3DRMFrame::SetPosition and IDirect3DRMFrame::SetOrientation methods. The IDirect3DRMFrame::SetRotation method imparts a spin to the frame to which the sphere will be added.

```
//////////////////////////////////////////////////////////////////
//
// SetMyPositions
// Set the positions and orientations of the light, camera, and
// world frames. Establish a rotation for the globe.
//
//////////////////////////////////////////////////////////////////

void SetMyPositions(LPDIRECT3DRMFRAME lpScene,
    LPDIRECT3DRMFRAME lpCamera, LPDIRECT3DRMFRAME lpLightFrame1,
    LPDIRECT3DRMFRAME lpWorld_frame)
{

    lpLightFrame1->lpVtbl->SetPosition(lpLightFrame1, lpScene,
        D3DVAL(2), D3DVAL(0.0), D3DVAL(22));

    lpCamera->lpVtbl->SetPosition(lpCamera, lpScene,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0));
    lpCamera->lpVtbl->SetOrientation(lpCamera, lpScene,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1),
        D3DVAL(0.0), D3DVAL(1), D3DVAL(0.0));

    lpWorld_frame->lpVtbl->SetPosition(lpWorld_frame, lpScene,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(15));
    lpWorld_frame->lpVtbl->SetOrientation(lpWorld_frame, lpScene,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1),
        D3DVAL(0.0), D3DVAL(1), D3DVAL(0.0));

    lpWorld_frame->lpVtbl->SetRotation(lpWorld_frame, lpScene,
        D3DVAL(0.0), D3DVAL(0.1), D3DVAL(0.0), D3DVAL(0.05));
}
```

The MakeMyMesh Function

The MyScene function calls the MakeMyMesh function to load and set the spherical mesh used in Helworld.c. MakeMyMesh calls the IDirect3DRM::CreateMeshBuilder method to create an IDirect3DRMMeshBuilder interface. Then it calls the IDirect3DRMMeshBuilder::Load, IDirect3DRMMeshBuilder::Scale, and IDirect3DRMMeshBuilder::SetColorRGB methods to prepare the mesh represented by the Sphere3.x file. (The Sphere3.x file is included in the DirectX SDK, as part of the media provided for use with the sample code.)

```
//////////////////////////////////////////////////////////////////
//
// MakeMyMesh
// Create MeshBuilder object, load, scale, and color the mesh.
//
//////////////////////////////////////////////////////////////////

void MakeMyMesh(LPDIRECT3DRMMESHBUILDER * lplpSphere3_builder)
{
    lpD3DRM->lpVtbl->CreateMeshBuilder(lpD3DRM, lplpSphere3_builder);

    (*lplpSphere3_builder)->lpVtbl->Load(*lplpSphere3_builder,
        "sphere3.x", NULL, D3DRMLOAD_FROMFILE, NULL, NULL);

    (*lplpSphere3_builder)->lpVtbl->Scale(*lplpSphere3_builder,
        D3DVAL(2), D3DVAL(2), D3DVAL(2));

    // Set sphere to white to avoid unexpected texture-blending results.

    (*lplpSphere3_builder)->lpVtbl->SetColorRGB(*lplpSphere3_builder,
        D3DVAL(1), D3DVAL(1), D3DVAL(1));
}
```

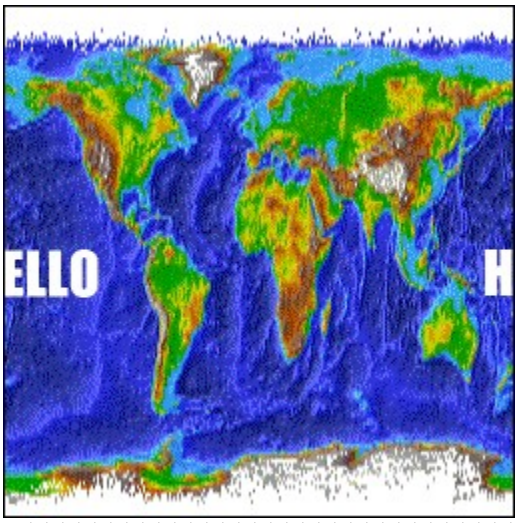
The MakeMyWrap Function

The MyScene function calls the MakeMyWrap function to create and apply texture coordinates to the sphere loaded by the MakeMyMesh function. MakeMyWrap calls the IDirect3DRMMeshBuilder::GetBox method to retrieve the bounding box that contains the sphere, and uses the dimensions of that bounding box in a call to the IDirect3DRM::CreateWrap method, which creates a cylindrical texture wrap and retrieves the IDirect3DRMWrap interface. A call to the IDirect3DRMWrap::Apply method applies the texture coordinates to the sphere.

```
//////////////////////////////////////////////////////////////////
//
// MakeMyWrap
// Creates and applies wrap for texture.
//
//////////////////////////////////////////////////////////////////

void MakeMyWrap(LPDIRECT3DRMMESHBUILDER sphere3_builder,
                LPDIRECT3DRMWRAP * lpWrap)
{
    D3DVALUE miny, maxy, height;
    D3DRMBOX box;

    sphere3_builder->lpVtbl->GetBox(sphere3_builder, &box);

    maxy = box.max.y;
    miny = box.min.y;
    height = maxy - miny;

    lpD3DRM->lpVtbl->CreateWrap
        (lpD3DRM, D3DRMWRAP_CYLINDER, NULL,
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(0.0),
        D3DVAL(0.0), D3DVAL(1.0), D3DVAL(0.0),
        D3DVAL(0.0), D3DVAL(0.0), D3DVAL(1.0),
        D3DVAL(0.0), D3DDivide(miny, height),
        D3DVAL(1.0), D3DDivide(-D3DVAL(1.0), height),
        lpWrap);

    (*lpWrap)->lpVtbl->Apply(*lpWrap, (LPDIRECT3DRMOBJECT)
        sphere3_builder);

}
```

The AddMyTexture Function

The MyScene function calls the AddMyTexture function to load a texture and associate it with the sphere. AddMyTexture calls the IDirect3DRM::LoadTexture method to load a bitmap called tutor.bmp, and then it calls the IDirect3DRMMeshBuilder::SetTexture method to associate the bitmap with the sphere.




```
////////////////////////////////////////////////////////////////////////
//
// AddMyTexture
// Creates and applies wrap for texture.
//
////////////////////////////////////////////////////////////////////////

void AddMyTexture(LPDIRECT3DRMMESHBUILDER lpSphere3_builder,
    LPDIRECT3DRMTEXTURE * lplpTex)
{
    lpD3DRM->lpVtbl->LoadTexture(lpD3DRM, "tutor.bmp", lplpTex);

    // If you need a color depth other than the default (16),
    // call IDirect3DRMTexture::SetShades here.

    lpSphere3_builder->lpVtbl->SetTexture(lpSphere3_builder, *lplpTex);

}
```

Cleaning Up

Helworld.c calls the CleanUp function when it receives a WM_DESTROY message or after several consecutive unsuccessful attempts to call the RenderLoop function.

```
//////////////////////////////////////////////////////////////////////
//
// CleanUp
// Release all D3DRM objects and set the bQuit flag.
//
//////////////////////////////////////////////////////////////////////

void
CleanUp(void)
{
    myglobs.bInitialized = FALSE;
    myglobs.scene->lpVtbl->Release(myglobs.scene);
    myglobs.camera->lpVtbl->Release(myglobs.camera);
    myglobs.view->lpVtbl->Release(myglobs.view);
    myglobs.dev->lpVtbl->Release(myglobs.dev);
    lpD3DRM->lpVtbl->Release(lpD3DRM);
    lpDDClipper->lpVtbl->Release(lpDDClipper);

    myglobs.bQuit = TRUE;
}
```

Direct3DRMCreate

Creates an instance of a Direct3DRM object.

HRESULT Direct3DRMCreate(
    LPDIRECT3DRM FAR * *lplpD3DRM*
    );

Parameters

lplpD3DRM
Address of a pointer that will be initialized with a valid Direct3DRM pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

Direct3DRMObject

D3DRMColorGetAlpha

Retrieves the alpha component of a color.

D3DVALUE D3DRMColorGetAlpha(
    D3DCOLOR *d3drmc*
    );

Parameters

d3drmc
Color from which the alpha component is retrieved.

Return Values

Returns the alpha value if successful, or zero otherwise.

See Also

D3DRMColorGetBlue, D3DRMColorGetGreen, D3DRMColorGetRed

D3DRMColorGetBlue

Retrieves the blue component of a color.

D3DVALUE D3DRMColorGetBlue(
    D3DCOLOR *d3drmc*
    );

Parameters

d3drmc
Color from which the blue component is retrieved.

Return Values

Returns the blue value if successful, or zero otherwise.

See Also

D3DRMColorGetAlpha, D3DRMColorGetGreen, D3DRMColorGetRed

D3DRMColorGetGreen

Retrieves the green component of a color.

D3DVALUE D3DRMColorGetGreen(
    D3DCOLOR *d3drmc*
    );

Parameters

d3drmc
Color from which the green component is retrieved.

Return Values

Returns the green value if successful, or zero otherwise.

See Also

D3DRMColorGetAlpha, D3DRMColorGetBlue, D3DRMColorGetRed

D3DRMColorGetRed

Retrieves the red component of a color.

D3DVALUE D3DRMColorGetRed(
    D3DCOLOR *d3drmc*
    );

Parameters

d3drmc
Color from which the red component is retrieved.

Return Values

Returns the red value if successful, or zero otherwise.

See Also

D3DRMColorGetAlpha, D3DRMColorGetBlue, D3DRMColorGetGreen

D3DRMCreateColorRGB

Creates an RGB color from supplied red, green, and blue components.

D3DCOLOR D3DRMCreateColorRGB(
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

red, *green*, and *blue*
Components of the RGB color.

Return Values

Returns the new RGB value if successful, or zero otherwise.

See Also

D3DRMCreateColorRGBA

D3DRMCreateColorRGBA

Creates an RGBA color from supplied red, green, blue, and alpha components.

D3DCOLOR D3DRMCreateColorRGBA(
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*,
D3DVALUE *alpha*
);

Parameters

red, *green*, *blue*, and *alpha*
Components of the RGBA color.

Return Values

Returns the new RGBA value if successful, or zero otherwise.

See Also

D3DRMCreateColorRGB

D3DRMFREEFUNCTION

Frees memory. This function is application-defined.

```
typedef VOID (*D3DRMFREEFUNCTION)(LPVOID lpArg);
typedef D3DRMFREEFUNCTION *LPD3DRMFREEFUNCTION;
```

Parameters

lpArg
Address of application-defined data.

Return Values

No return value.

Remarks

Applications might define their own memory-freeing function if the standard C run-time routines do not meet their requirements.

D3DRMMALLOCFUNCTION

Allocates memory. This function is application-defined.

```
typedef LPVOID (*D3DRMMALLOCFUNCTION)(DWORD dwSize);
typedef D3DRMMALLOCFUNCTION *LPD3DRMMALLOCFUNCTION;
```

Parameters

dwSize

Specifies the size, in bytes, of the memory that will be allocated.

Return Values

Returns the address of the allocated memory if successful, or zero otherwise.

Remarks

Applications might define their own memory-allocation function if the standard C run-time routines do not meet their requirements.

D3DRMMatrixFromQuaternion

Calculates the matrix for the rotation that a unit quaternion represents.

void D3DRMMatrixFromQuaternion (
D3DRMMATRIX4D *mat*,
LPD3DRMQUATERNION *lpquat*
);

Parameters

mat
Address that will contain the calculated matrix when the function returns. (The D3DRMMATRIX4D type is an array.)

lpquat
Address of the D3DRMQUATERNION structure.

Return Values

No return value.

D3DRMQuaternionFromRotation

Retrieves a unit quaternion that represents a rotation of a specified number of radians around the given axis.

LPD3DRMQUATERNION D3DRMQuaternionFromRotation(
    LPD3DRMQUATERNION *lpquat*,
    LPD3DVECTOR *lpv*,
    D3DVALUE *theta*
    );

Parameters

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.

Return Values

Returns the address of the unit quaternion that was passed as the first parameter if successful, or zero otherwise.

D3DRMQuaternionMultiply

Calculates the product of two quaternion structures.

LPD3DRMQUATERNION D3DRMQuaternionMultiply(
LPD3DRMQUATERNION *lpq*,
LPD3DRMQUATERNION *lpa*,
LPD3DRMQUATERNION *lpb*
);

Parameters

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.

Return Values

Returns the address of the quaternion that was passed as the first parameter if successful, or zero otherwise.

D3DRMQuaternionSlerp

Interpolates between two quaternion structures, using spherical linear interpolation.

LPD3DRMQUATERNION D3DRMQuaternionSlerp(
    LPD3DRMQUATERNION *lpq*,
    LPD3DRMQUATERNION *lpa*,
    LPD3DRMQUATERNION *lpb*,
    D3DVALUE *alpha*
    );

Parameters

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*.

Return Values

Returns the address of the quaternion that was passed as the first parameter if successful, or zero otherwise.

D3DRMREALLOCFUNCTION

Reallocates memory. This function is application-defined.

```
typedef LPVOID (*D3DRMREALLOCFUNCTION)(LPVOID lpArg,
    DWORD dwSize);
typedef D3DRMREALLOCFUNCTION *LPD3DRMREALLOCFUNCTION;
```

Parameters

lpArg
Address of application-defined data.

dwSize
Size, in bytes, of the reallocated memory.

Return Values

Returns an address of the reallocated memory if successful, or zero otherwise.

Remarks

Applications may define their own memory-reallocation function if the standard C run-time routines do not meet their requirements.

D3DRMVectorAdd

Adds two vectors.

LPD3DVECTOR D3DRMVectorAdd(
    LPD3DVECTOR *lpd*,
    LPD3DVECTOR *lps1*,
    LPD3DVECTOR *lps2*
    );

Parameters

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.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMVectorCrossProduct

Calculates the cross product of the two vector arguments.

LPD3DVECTOR D3DRMVectorCrossProduct(
LPD3DVECTOR *lpd*,
LPD3DVECTOR *lps1*,
LPD3DVECTOR *lps2*
);

Parameters

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.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMVectorDotProduct

Returns the vector dot product.

D3DVALUE D3DRMVectorDotProduct(
LPD3DVECTOR *lps1*,
LPD3DVECTOR *lps2*
);

Parameters

lps1 and *lps2*
Addresses of the **D3DVECTOR** structures from which the dot product is produced.

Return Values

Returns the result of the dot product if successful, or zero otherwise.

D3DRMVectorModulus

Returns the length of a vector according to the following formula:

$length = \sqrt{x^2 + y^2 + z^2}$

D3DVALUE D3DRMVectorModulus(
LPD3DVECTOR *lpv*
);

Parameters

lpv

Address of the **D3DVECTOR** structure whose length is returned.

Return Values

Returns the length of the **D3DVECTOR** structure if successful, or zero otherwise.

D3DRMVectorNormalize

Scales a vector so that its modulus is 1.

LPD3DVECTOR D3DRMVectorNormalize(
    LPD3DVECTOR *lpv*
    );

Parameters

lpv
Address of a **D3DVECTOR** structure that will contain the result of the scaling operation.

Return Values

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.

D3DRMVectorRandom

Returns a random unit vector.

LPD3DVECTOR D3DRMVectorRandom(
    LPD3DVECTOR *lpd*
    );

Parameters

lpd
Address of a **D3DVECTOR** structure that will contain a random unit vector.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMVectorReflect

Reflects a ray about a given normal.

LPD3DVECTOR D3DRMVectorReflect(
    LPD3DVECTOR *lpd*,
    LPD3DVECTOR *lpRay*,
    LPD3DVECTOR *lpNorm*
    );

Parameters

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.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMVectorRotate

Rotates a vector around a given axis. Returns a normalized vector if successful.

LPD3DVECTOR D3DRMVectorRotate(
    LPD3DVECTOR *lpr*,
    LPD3DVECTOR *lpv*,
    LPD3DVECTOR *lpaxis*,
    D3DVALUE *theta*
    );

Parameters

lpr
Address of a **D3DVECTOR** structure that will contain the normalized 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.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise. This vector is normalized.

D3DRMVectorScale

Scales a vector uniformly in all three axes.

LPD3DVECTOR D3DRMVectorScale(
    LPD3DVECTOR *lpd*,
    LPD3DVECTOR *lps*,
    D3DVALUE *factor*
    );

Parameters

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.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMVectorSubtract

Subtracts two vectors.

LPD3DVECTOR D3DRMVectorSubtract(
    LPD3DVECTOR *lpd*,
    LPD3DVECTOR *lps1*,
    LPD3DVECTOR *lps2*
    );

Parameters

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*.

Return Values

Returns the address of the vector that was passed as the first parameter if successful, or zero otherwise.

D3DRMDEVICEPALETTECALLBACK

Enumerates palette entries. This callback function is application-defined.

```
void (
        *D3DRMDEVICEPALETTECALLBACK
        )
(
        LPDIRECT3DRMDEVICE lpDirect3DRMDev,
        LPVOID lpArg,
        DWORD dwIndex,
        LONG red,
        LONG green,
        LONG blue
        );
```

Parameters

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.

Return Values

No return value.

Remarks

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

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.

```
void (
    *D3DRMFRAMEMOVECALLBACK
    )(
    LPDIRECT3DRMFRAME lpD3DRMFrame,
    LPVOID lpArg,
    D3DVALUE delta
    );
```

Parameters

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.

Return Values

No return value.

Remarks

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

Loads objects named in a call to the IDirect3DRM::Load method. This callback function is application-defined.

```
void (
    *D3DRMLOADCALLBACK
    )(
    LPDIRECT3DRMOBJECT lpObject,
    REFIID ObjectGuid,
    LPVOID lpArg
    );
```

Parameters

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.

Return Values

No return value.

Remarks

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.

See Also

IDirect3DRM::Load

D3DRMLOADTEXTURECALLBACK

Loads texture maps from a file or resource named in a call to one of the **Load** methods. This callback function is application-defined.

HRESULT (
**D3DRMLOADTEXTURECALLBACK*
)(
char **tex_name*,
void **lpArg*,
LPDIRECT3DRMTEXTURE **lpD3DRMTex*
);

Parameters

tex_name
Address of a string containing the name of the texture.

lpArg
Address of application-specific data.

lpD3DRMTex
Address of the Direct3DRMTexture object.

Return Values

Should return D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRM::Load, IDirect3DRMAnimationSet::Load, IDirect3DRMFrame::Load, IDirect3DRMMeshBuilder::Load

D3DRMOBJECTCALLBACK

Enumerates objects in response to a call to the IDirect3DRM::EnumerateObjects method. This callback function is application-defined.

```
void (
    *D3DRMOBJECTCALLBACK
    )(
    LPDIRECT3DRMOBJECT lpD3DRMobj,
    LPVOID lpArg
    );
```

Parameters

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.

Return Values

No return value.

Remarks

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.

See Also

IDirect3DRM::EnumerateObjects

D3DRMUPDATECALLBACK

Alerts the application whenever the device changes. This callback function is application-defined.

```
void (
    *D3DRMUPDATECALLBACK
    )(
    LPDIRECT3DRMDEVICE lpobj,
    LPVOID lpArg,
    int iRectCount,
    LPD3DRECT d3dRectUpdate
    );
```

Parameters

lpobj
Address of the Direct3DRMDevice object to which this callback function applies.

lpArg
Address of application-defined data passed to this callback function.

iRectCount
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.

Return Values

No return value.

Remarks

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.

See Also

IDirect3DRMDevice::AddUpdateCallback, IDirect3DRMDevice::DeleteUpdateCallback, IDirect3DRMDevice::Update

D3DRMUSERVISUALCALLBACK

Alerts an application that supplies user-visual objects that it should execute the execute buffer. This function is application-defined.

```
int (
        *D3DRMUSERVISUALCALLBACK
        )(
        LPDIRECT3DRMUSERVISUAL lpD3DRMUV,
        LPVOID lpArg,
        D3DRMUSERVISUALREASON lpD3DRMUVreason,
        LPDIRECT3DRMDEVICE lpD3DRMDev,
        LPDIRECT3DRMVIEWPORT lpD3DRMview
        );
```

Parameters

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.

Return Values

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.

Remarks

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.

See Also

IDirect3DRMUserVisual::Init

D3DRMWRAPCALLBACK

This callback function is not supported.

void (
**D3DRMWRAPCALLBACK*
)(
LPD3DVECTOR *lpD3DVector*,
int* *lpU*,
int* *lpV*,
LPD3DVECTOR *lpD3DRMVA*,
LPD3DVECTOR *lpD3DRMVB*,
LPVOID *lpArg*
);

IDirect3DRM 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
IDirect3DRMObjectArray
IDirect3DRMPickedArray
IDirect3DRMPicked2Array
IDirect3DRMViewportArray
IDirect3DRMVisualArray

IDirect3DRMArray

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 section is a reference to the methods of this interface.

The **IDirect3DRMArray** interface supports the GetSize method.

The **IDirect3DRMArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. This interface supports the following three methods:

AddRef

QueryInterface

Release

IDirect3DRMArray::GetSize

Retrieves the size, in objects, of the Direct3DRMArray object.

DWORD GetSize( );

Return Values

Returns the size.

IDirect3DRMDeviceArray

Applications use the methods of the **IDirect3DRMDeviceArray** interface to organize device objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMDevice, IDirect3DRMDevice2, and IDirect3DRMDeviceArray Interfaces.

The **IDirect3DRMDeviceArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMDeviceArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMDeviceArray object is obtained by calling the IDirect3DRM::GetDevices method.

IDirect3DRMDeviceArray::GetElement

Retrieves a specified element in a Direct3DRMDeviceArray object.

HRESULT GetElement(
DWORD *index*,
LPDIRECT3DRMDEVICE * *lplpD3DRMDevice*
);

Parameters

index
Element in the array.

lplpD3DRMDevice
Address that will be filled with a pointer to an IDirect3DRMDevice interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDeviceArray::GetSize

Retrieves the number of elements contained in a Direct3DRMDeviceArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMFaceArray

Applications use the methods of the **IDirect3DRMFaceArray** interface to organize faces in a mesh. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMFace and IDirect3DRMFaceArray Interfaces.

The **IDirect3DRMFaceArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMFaceArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMFaceArray object is obtained by calling the IDirect3DRMMeshBuilder::GetFaces method.

IDirect3DRMFaceArray::GetElement

Retrieves a specified element in a Direct3DRMFaceArray object.

HRESULT GetElement(
    DWORD *index*,
    LPDIRECT3DRMFACE * *lplpD3DRMFace*
    );

Parameters

index
Element in the array.

lplpD3DRMFace
Address that will be filled with a pointer to an IDirect3DRMFace interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFaceArray::GetSize

Retrieves the number of elements contained in a Direct3DRMFaceArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMFrameArray

Applications use the methods of the **IDirect3DRMFrameArray** interface to organize frame objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMFrame, IDirect3DRMFrame2, and IDirect3DRMFrameArray Interfaces.

The **IDirect3DRMFrameArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMFrameArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMFrameArray object is obtained by calling the IDirect3DRMPickedArray::GetPick, IDirect3DRMPicked2Array::GetPick, or IDirect3DRMFrame::GetChildren method.

IDirect3DRMFrameArray::GetElement

Retrieves a specified element in a Direct3DRMFrameArray object.

HRESULT GetElement(
    DWORD *index*,
    LPDIRECT3DRMFRAME * *lplpD3DRMFrame*
    );

Parameters

index
Element in the array.

lplpD3DRMFrame
Address that will be filled with a pointer to an IDirect3DRMFrame interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrameArray::GetSize

Retrieves the number of elements contained in a Direct3DRMFrameArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMLightArray

Applications use the methods of the **IDirect3DRMLightArray** interface to organize light objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMLight and IDirect3DRMLightArray Interfaces.

The **IDirect3DRMLightArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMLightArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMLightArray object is obtained by calling the IDirect3DRMFrame::GetLights method.

IDirect3DRMLightArray::GetElement

Retrieves a specified element in a Direct3DRMLightArray object.

HRESULT GetElement(
DWORD *index*,
LPDIRECT3DRMLIGHT * *lplpD3DRMLight*
);

Parameters

index
Element in the array.

lplpD3DRMLight
Address that will be filled with a pointer to an IDirect3DRMLight interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMLightArray::GetSize

Retrieves the number of elements contained in a Direct3DRMLightArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMObjectArray

Applications use the methods of the **IDirect3DRMObjectArray** interface to organize Direct3DRMObject objects. This section is a reference to the methods of this interface. For a conceptual overview, see Direct3DRMObject.

The **IDirect3DRMObjectArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMObjectArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMObjectArray object is obtained by calling the IDirect3DRMInterpolator::GetAttachedObjects method.

IDirect3DRMObjectArray::GetElement

Retrieves a specified element in a Direct3DRMObjectArray object.

HRESULT GetElement(
DWORD *index*,
LPDIRECT3DRMOBJECT * *lplpD3DRMObject*
);

Parameters

index
Element in the array.

lplpD3DRMObject
Address that will be filled with a pointer to an IDirect3DRMObject interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObjectArray::GetSize

Retrieves the number of elements contained in a Direct3DRMObjectArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMPickedArray

Applications use the methods of the **IDirect3DRMPickedArray** interface to organize pick objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMPickedArray and IDirect3DRMPicked2Array Interfaces.

The **IDirect3DRMPickedArray** interface supports the following methods:

GetPick
GetSize

The **IDirect3DRMPickedArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMPickedArray object is obtained by calling the IDirect3DRMViewport::Pick method.

IDirect3DRMPickedArray::GetPick

Retrieves the Direct3DRMVisual and Direct3DRMFrame objects intersected by the specified pick.

HRESULT GetPick(
DWORD *index*,
LPDIRECT3DRMVISUAL * *lplpVisual*,
LPDIRECT3DRMFRAMEARRAY * *lplpFrameArray*,
LPD3DRMPICKDESC *lpD3DRMPickDesc*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMViewport::Pick

IDirect3DRMPickedArray::GetSize

Retrieves the number of elements contained in a Direct3DRMPickedArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMPicked2Array

Applications use the methods of the **IDirect3DRMPicked2Array** interface to organize pick objects and return more information about the pick objects. The D3DRMPICKDESC2 structure returned by IDirect3DRMPicked2Array::GetPick contains the face and group identifiers, pick position, horizontal and vertical texture coordinates for the vertex, vertex normal, and color of the intersected objects. A pointer to this interface pointer is returned in the IDirect3DRMFrame2::RayPick method during ray picks.

This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMPickedArray and IDirect3DRMPicked2Array Interfaces.

The **IDirect3DRMPicked2Array** interface supports the following methods:

GetPick
GetSize

The **IDirect3DRMPicked2Array** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMPicked2Array object is obtained by calling the IDirect3DRMFrame2::RayPick method.

IDirect3DRMPicked2Array::GetPick

Retrieves the Direct3DRMVisual and Direct3DRMFrame objects intersected by the specified pick.

HRESULT GetPick(
DWORD *index*,
LPDIRECT3DRMVISUAL * *lplpVisual*,
LPDIRECT3DRMFRAMEARRAY * *lplpFrameArray*,
LPD3DRMPICKDESC2 *lpD3DRMPickDesc2*
);

Parameters

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 D3DRMPICKDESC2 structure specifying the face and group identifiers, pick position, horizontal and vertical texture coordinates for the vertex, vertex normal, and color of the intersected objects.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::RayPick

IDirect3DRMPicked2Array::GetSize

Retrieves the number of elements contained in a Direct3DRMPicked2Array object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMViewportArray

Applications use the methods of the **IDirect3DRMViewportArray** interface to organize viewport objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMViewport and IDirect3DRMViewportArray Interface.

The **IDirect3DRMViewportArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMViewportArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMViewportArray object is obtained by calling the IDirect3DRM::CreateFrame method.

IDirect3DRMViewportArray::GetElement

Retrieves a specified element in a Direct3DRMViewportArray object.

HRESULT GetElement(
DWORD *index*,
LPDIRECT3DRMVIEWPORT * *lplpD3DRMViewport*
);

Parameters

index
Element in the array.

lplpD3DRMViewport
Address that will be filled with a pointer to an IDirect3DRMViewport interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewportArray::GetSize

Retrieves the number of elements contained in a Direct3DRMViewportArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRMVisualArray

Applications use the methods of the **IDirect3DRMVisualArray** interface to organize groups of visual objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMVisual and IDirect3DRMVisualArray Interfaces.

The **IDirect3DRMVisualArray** interface supports the following methods:

GetElement
GetSize

The **IDirect3DRMVisualArray** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The Direct3DRMVisualArray object is obtained by calling the IDirect3DRMFrame::GetVisuals method.

IDirect3DRMVisualArray::GetElement

Retrieves a specified element in a Direct3DRMVisualArray object.

HRESULT GetElement(
DWORD *index*,
LPDIRECT3DRMVISUAL * *lplpD3DRMVisual*
);

Parameters

index
Element in the array.

lplpD3DRMVisual
Address that will be filled with a pointer to an **IDirect3DRMVisual** interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMVisualArray::GetSize

Retrieves the number of elements contained in a Direct3DRMVisualArray object.

DWORD GetSize( );

Return Values

Returns the number of elements.

IDirect3DRM

Applications use the methods of the **IDirect3DRM** interface to create Direct3DRM objects and work with system-level variables. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRM and IDirect3DRM2 Interfaces.

The methods of the **IDirect3DRM** interface can be organized into the following groups:

Animation	CreateAnimation CreateAnimationSet
Devices	CreateDevice CreateDeviceFromClipper CreateDeviceFromD3D CreateDeviceFromSurface GetDevices
Enumeration	EnumerateObjects
Faces	CreateFace
Frames	CreateFrame
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

The **IDirect3DRM** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

The **IDirect3DRM** COM interface is created by calling the Direct3DRMCreate function.

IDirect3DRM::AddSearchPath

Adds a list of directories to the end of the current file search path.

HRESULT AddSearchPath(
    LPCSTR *lpPath*
    );

Parameters

lpPath
Address of a null-terminated string specifying the path to add to the current search path.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

For Windows, the path should be a list of directories separated by semicolons (;).

See Also

IDirect3DRM::SetSearchPath

IDirect3DRM::CreateAnimation

Creates an empty Direct3DRMAnimation object.

HRESULT CreateAnimation(
LPDIRECT3DRMANIMATION * *lplpD3DRMAnimation*
);

Parameters

lplpD3DRMAnimation
Address that will be filled with a pointer to an IDirect3DRMAnimation interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateAnimationSet

Creates an empty Direct3DRMAnimationSet object.

HRESULT CreateAnimationSet (
    LPDIRECT3DRMANIMATIONSET * *lplpD3DRMAnimationSet*
    );

Parameters

lplpD3DRMAnimationSet
Address that will be filled with a pointer to an IDirect3DRMAnimationSet interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateDevice

Not implemented on the Windows platform.

HRESULT CreateDevice(
DWORD *dwWidth*,
DWORD *dwHeight*,
LPDIRECT3DRMDEVICE* *lplpD3DRMDevice*
);

IDirect3DRM::CreateDeviceFromClipper

Creates a Direct3DRM Windows device by using a specified DirectDrawClipper object.

HRESULT CreateDeviceFromClipper(
LPDIRECTDRAWCLIPPER *lpDDClipper*,
LPGUID *lpGUID*,
int *width*,
int *height*,
LPDIRECT3DRMDEVICE * *lplpD3DRMDevice*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Creates a Direct3DRM Windows device by using specified Direct3D objects.

HRESULT CreateDeviceFromD3D(
LPDIRECT3D *lpD3D*,
LPDIRECT3DDEVICE *lpD3DDev*,
LPDIRECT3DRMDEVICE * *lplpD3DRMDevice*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateDeviceFromSurface

Creates a Windows device for rendering from the specified DirectDraw surfaces.

HRESULT CreateDeviceFromSurface(
LPGUID *lpGUID*,
LPDIRECTDRAW *lpDD*,
LPDIRECTDRAWSURFACE *lpDDSBack*,
LPDIRECT3DRMDEVICE * *lplpD3DRMDevice*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateFace

Creates an instance of the IDirect3DRMFace interface.

HRESULT CreateFace(
LPDIRECT3DRMFACE * *lplpd3drmFace*
);

Parameters

lplpd3drmFace
Address that will be filled with a pointer to an IDirect3DRMFace interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateFrame

Creates a new child frame of the given parent frame.

HRESULT CreateFrame(
LPDIRECT3DRMFRAME *lpD3DRMFrame*,
LPDIRECT3DRMFRAME* *lplpD3DRMFrame*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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 associate it with a parent frame later by using the IDirect3DRMFrame::AddChild method.

See Also

IDirect3DRMFrame::AddChild

IDirect3DRM::CreateLight

Creates a new light source with the given type and color.

HRESULT CreateLight(
D3DRMLIGHTTYPE *d3drmltLightType*,
D3DCOLOR *cColor*,
LPDIRECT3DRMLIGHT* *lplpD3DRMLight*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateLightRGB

Creates a new light source with the given type and color.

HRESULT CreateLightRGB(
D3DRMLIGHTTYPE *ltLightType*,
D3DVALUE *vRed*,
D3DVALUE *vGreen*,
D3DVALUE *vBlue*,
LPDIRECT3DRMLIGHT* *lplpD3DRMLight*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateMaterial

Creates a material with the given specular property.

HRESULT CreateMaterial(
    D3DVALUE *vPower*,
    LPDIRECT3DRMMATERIAL * *lplpD3DRMMaterial*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateMesh

Creates a new mesh object with no faces. The mesh is not visible until it is added to a frame.

HRESULT CreateMesh(
    LPDIRECT3DRMMESH* *lplpD3DRMMesh*
    );

Parameters

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateMeshBuilder

Creates a new mesh builder object.

HRESULT CreateMeshBuilder(
    LPDIRECT3DRMMESHBUILDER* *lplpD3DRMMeshBuilder*
    );

Parameters

lplpD3DRMMeshBuilder
Address that will be filled with a pointer to an IDirect3DRMMeshBuilder interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateObject

Creates a new object without initializing the object.

HRESULT CreateObject(
REFCLSID *rclsid*,
LPUNKNOWN *pUnkOuter*,
REFIID *riid*,
LPVOID FAR* *ppv*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

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.

HRESULT CreateShadow(
LPDIRECT3DRMVISUAL *lpVisual*,
LPDIRECT3DRMLIGHT *lpLight*,
D3DVALUE *px*,
D3DVALUE *py*,
D3DVALUE *pz*,
D3DVALUE *nx*,
D3DVALUE *ny*,
D3DVALUE *nz*,
LPDIRECT3DRMVISUAL * *lplpShadow*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateTexture

Creates a texture from an image in memory.

HRESULT CreateTexture(
    LPD3DRMIMAGE *lpImage*,
    LPDIRECT3DRMTEXTURE* *lplpD3DRMTexture*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Creates a texture from a specified DirectDraw surface.

```
HRESULT CreateTextureFromSurface(
    LPDIRECTDRAWSURFACE lpDDS,
    LPDIRECT3DRMTEXTURE * lplpD3DRMTexture
    );
```

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateUserVisual

Creates an application-defined visual object, which can then be added to a scene and rendered by using an application-defined handler.

HRESULT CreateUserVisual(
D3DRMUSERVISUALCALLBACK *fn*,
LPVOID *lpArg*,
LPDIRECT3DRMUSERVISUAL * *lplpD3DRMUV*
);

Parameters

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.

Return Values

Should return D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::CreateViewport

Creates a viewport on a device with device coordinates (*dwXPos*, *dwYPos*) to (*dwXPos* + *dwWidth*, *dwYPos* + *dwHeight*).

HRESULT CreateViewport(
LPDIRECT3DRMDEVICE *lpDev*,
LPDIRECT3DRMFRAME *lpCamera*,
DWORD *dwXPos*,
DWORD *dwYPos*,
DWORD *dwWidth*,
DWORD *dwHeight*,
LPDIRECT3DRMVIEWPORT* *lplpD3DRMViewport*
);

Parameters

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. The viewport size cannot be larger than the physical device or the method will fail.

lplpD3DRMViewport
Address that will be filled with a pointer to an IDirect3DRMViewport interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. If the viewport size is larger than the physical device, returns D3DRMERR_BADVALUE. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The viewport displays objects in the scene that contains the camera, with the view direction and up vector taken from the camera. The viewport size cannot be larger than the physical device.

IDirect3DRM::CreateWrap

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.

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*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMWrap

IDirect3DRM::EnumerateObjects

Calls the callback function specified by the *func* parameter on each of the active Direct3DRM objects.

HRESULT EnumerateObjects(
D3DRMOBJECTCALLBACK *func*,
LPVOID *lpArg*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::GetDevices

Returns all the Direct3DRM devices that have been created in the system.

HRESULT GetDevices(
    LPDIRECT3DRMDEVICEARRAY* *lplpDevArray*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::GetNamedObject

Finds a Direct3DRMObject, given its name.

HRESULT GetNamedObject(
const char * *lpName*,
LPDIRECT3DRMOBJECT* *lplpD3DRMObject*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

If the system does not find an object with the name specified in the *lpName* parameter, this method returns D3DRM_OK but the *lplpD3DRMObject* parameter is NULL.

IDirect3DRM::GetSearchPath

Returns the current search path. For Windows, the path is a list of directories separated by semicolons (;).

HRESULT GetSearchPath(
DWORD * *lpdwSize***,**
LPSTR *lpszPath*
);

Parameters

lpdwSize

Pointer to a DWORD. Should contain the length of *lpszPath* when **GetSearchPath** is called. On return, contains the length of the string in *lpszPath* containing the current search path. This parameter cannot be NULL.

lpszPath

On return, contains a pointer to a null-terminated string specifying the search path. If *lpszPath* equals NULL when **GetSearchPath** is called, the method returns the length of the current string in the location pointed to by the *lpdwSize* parameter.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRM::SetSearchPath

IDirect3DRM::Load

Loads an object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
LPIID * *lplpGUIDs*,
DWORD *dwcGUIDs*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADCALLBACK *d3drmLoadProc*,
LPVOID *lpArgLP*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpArgLTP*,
LPDIRECT3DRMFRAME *lpParentFrame*
);

Parameters

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_IDirect3DRMMeshBuilder and IID_IDirect3DRMAnimationSet, this method loads all the animation sets and mesh-builder objects. Possible GUIDs must be one or more of the following: IID_IDirect3DRMMeshBuilder, IID_IDirect3DRMAnimationSet, IID_IDirect3DRMAnimation, and IID_IDirect3DRMFrame.

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 that require special formatting. This parameter can be NULL.

lpArgLTP
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

lpParentFrame
Address of a parent Direct3DRMFrame object. This argument only affects the loading of animation sets. When an animation that is loaded from an X file references an unparented frame in the X file, its parent is set to this parent frame argument. However, if you ask **Load** to load any frames in the X file, the parent frame argument will not be used as the parent frame for frames in the X file with no parent. That is, the parent frame argument is used only when you load animation sets. This value of this argument can be NULL.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method allows great flexibility in loading objects from DirectX files. Here is an example of how to use it:

```
#include <windows.h>
#include <stdio.h>
#include <initguid.h>
#include <d3drm.h>

LPDIRECT3DRM d3dApi;

int totalMesh = 0, totalAnim = 0, totalFrames = 0;

HRESULT loadTexture(LPSTR name, LPVOID lpArg, LPDIRECT3DRMTEXTURE *tex)
{
  return d3dApi->lpVtbl->LoadTexture(d3dApi, name, tex);
}

void loadCallback(LPDIRECT3DRMOBJECT lpObject, REFIID objGuid, LPVOID lpArg)
{
  if (IsEqualIID(objGuid, &IID_IDirect3DRMFrame)) {
    totalFrames ++;
    return;
  }

  if (IsEqualIID(objGuid, &IID_IDirect3DRMAnimationSet)) {
    totalAnim ++;
    return;
  }

  if (IsEqualIID(objGuid, &IID_IDirect3DRMMeshBuilder)) {
    totalMesh ++;
    return;
  }

  return;
}


BOOL loadObjects(LPSTR filename)
{
  LPGUID guids[] = { (LPGUID)&IID_IDirect3DRMMeshBuilder,
    (LPGUID)&IID_IDirect3DRMAnimationSet,
    (LPGUID)&IID_IDirect3DRMFrame };

  /* Tell the loader which objects you're interested in */

  if (FAILED(d3dApi->lpVtbl->Load(d3dApi, filename, NULL,
      guids, 3, D3DRMLOAD_FROMFILE,
      loadCallback, NULL, loadTexture, NULL, NULL)))
    return FALSE;

  printf("Total Frames loaded = %d\n", totalFrames);
  printf("Total Animation Sets loaded = %d\n", totalAnim);
  printf("Total Meshbuilders loaded = %d\n", totalMesh);
```

```
  return TRUE;
}

int main(int argc, char **argv)
{

  if (FAILED(Direct3DRMCreate(&d3dApi)))
    return FALSE;

  if (argc != 2) {
    fprintf(stderr, "usage: %s filename\n", argv[0]);
    return FALSE;
  }

  if (!loadObjects(argv[1])) return FALSE;

  return(0);
}
```

IDirect3DRM::LoadTexture

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.

HRESULT LoadTexture(
const char * *lpFileName*,
LPDIRECT3DRMTEXTURE* *lplpD3DRMTexture*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

LoadTexture checks whether the texture is in BMP or PPM format, which are the formats it knows how to load. If you want to load other formats, you can add code to the callback to load the image into a D3DRMIMAGE structure and then call IDirect3DRM::CreateTexture.

If you write your own texture callback, the **LoadTexture** call in the texture callback does not take a reference to the texture. For example:

```
HRESULT loadTextures(char *name, void *arg, LPDIRECT3DRMTEXTURE *tex)
{
    return lpD3DRM->LoadTexture(name, tex);
}
```

In this sample, no reference is taken for *tex*. If you want to keep a reference to each texture loaded by your texture callback, you should **AddRef** the texture. For example:

```
LPDIRECT3DRMTEXTURE *texarray;

HRESULT loadTextures(char *name, void *arg, LPDIRECT3DRMTEXTURE *tex)
{
  if (FAILED(lpD3DRM->LoadTexture(name, tex)) {
    return NULL;
  }

  texArray[current++] = tex;
  tex->AddRef();

  return tex;
}
```

IDirect3DRM::LoadTextureFromResource

Loads a texture from a specified resource.

HRESULT LoadTextureFromResource(
    HRSRC *rs*,
    LPDIRECT3DRMTEXTURE * *lplpD3DRMTexture*
    );

Parameters

rs

Handle of the resource.

lplpD3DRMTexture

Address of a pointer to be initialized with a valid Direct3DRMTexture object if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM::SetDefaultTextureColors

Sets the default colors to be used for a Direct3DRMTexture object.

HRESULT SetDefaultTextureColors(
    DWORD *dwColors*
    );

Parameters

dwColors
Number of colors.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Sets the default shades to be used for an Direct3DRMTexture object.

HRESULT SetDefaultTextureShades(
    DWORD *dwShades*
    );

Parameters

dwShades
Number of shades.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Sets the current file search path from a list of directories. For Windows, the path should be a list of directories separated by semicolons (;).

HRESULT SetSearchPath(
    LPCSTR *lpPath*
    );

Parameters

lpPath
Address of a null-terminated string specifying the path to set as the current search path.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRM::GetSearchPath

IDirect3DRM::Tick

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.

HRESULT Tick(
D3DVALUE *d3dvalTick*
);

Parameters

d3dvalTick

Velocity and rotation step for the IDirect3DRMFrame::SetRotation and IDirect3DRMFrame::SetVelocity methods.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You can implement this method by using other Retained-Mode methods to allow more flexibility in rendering a scene.

IDirect3DRM2

Applications use the methods of the **IDirect3DRM2** interface to create Direct3DRM2 objects and work with system-level variables.

Applications use the methods of the IDirect3DRMDevice2 and IDirect3DRMDevice interfaces to interact with the output device. While the **IDirect3DRMDevice** interface, when created from the IDirect3DRM interface, works with an **IDirect3DDevice** Immediate-Mode device, the **IDirect3DRMDevice2** interface, when created from the **IDirect3DRM2** interface or initialized by the IDirect3DRMDevice2::InitFromClipper, IDirect3DRMDevice2::InitFromD3D2, or IDirect3DRMDevice2::InitFromSurface method, works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRM2 supports all the methods in **IDirect3DRM**. In addition, the IDirect3DRM2::CreateProgressiveMesh method had been added. The IDirect3DRM2::CreateDeviceFromSurface, IDirect3DRM2::CreateDeviceFromD3D, and IDirect3DRM2::CreateDeviceFromClipper methods all create a DIRECT3DRMDEVICE2 object. The IDirect3DRM2::CreateViewport method creates a viewport on a DirectDRMDevice2 object. The IDirect3DRM2::LoadTexture and IDirect3DRM2::LoadTextureFromResource methods load a Direct3DRMTexture2 object.

To access the **IDirect3DRM2** COM interface, create an IDirect3DRM object with Direct3DRMCreate, then query for **IDirect3DRM2** from IDirect3DRM.

For a conceptual overview, see IDirect3DRM and IDirect3DRM2 Interfaces.

The methods of the **IDirect3DRM2** interface can be organized into the following groups:

Animation	CreateAnimation CreateAnimationSet
Devices	CreateDevice CreateDeviceFromClipper CreateDeviceFromD3D CreateDeviceFromSurface GetDevices
Enumeration	EnumerateObjects
Faces	CreateFace
Frames	CreateFrame
Lights	CreateLight CreateLightRGB
Materials	CreateMaterial
Meshes	CreateMesh CreateMeshBuilder
Miscellaneous	CreateObject CreateUserVisual GetNamedObject Load

	Tick
Progressive Meshes	CreateProgressiveMesh
Search paths	AddSearchPath GetSearchPath SetSearchPath
Shadows	CreateShadow
Textures	CreateTexture CreateTextureFromSurface LoadTexture LoadTextureFromResource SetDefaultTextureColors SetDefaultTextureShades
Viewports	CreateViewport
Wraps	CreateWrap

The **IDirect3DRM2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

IDirect3DRM2::AddSearchPath

Adds a list of directories to the end of the current file search path.

HRESULT AddSearchPath(
    LPCSTR *lpPath*
    );

Parameters

lpPath
Address of a null-terminated string specifying the path to add to the current search path.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

For Windows, the path should be a list of directories separated by semicolons (;).

See Also

IDirect3DRM2::SetSearchPath

IDirect3DRM2::CreateAnimation

Creates an empty Direct3DRMAnimation object.

HRESULT CreateAnimation(
    LPDIRECT3DRMANIMATION * *lplpD3DRMAnimation*
    );

Parameters

lplpD3DRMAnimation
Address that will be filled with a pointer to an IDirect3DRMAnimation interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateAnimationSet

Creates an empty Direct3DRMAnimationSet object.

HRESULT CreateAnimationSet (
    LPDIRECT3DRMANIMATIONSET * *lplpD3DRMAnimationSet*
    );

Parameters

lplpD3DRMAnimationSet
Address that will be filled with a pointer to an IDirect3DRMAnimationSet interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateDevice

Not implemented on the Windows platform.

HRESULT CreateDevice(
DWORD *dwWidth*,
DWORD *dwHeight*,
LPDIRECT3DRMDEVICE* *lplpD3DRMDevice*
);

IDirect3DRM2::CreateDeviceFromClipper

Creates a Direct3DRMDevice2 Windows device by using a specified DirectDrawClipper object. An **IDirect3DRMDevice2** interface works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

HRESULT CreateDeviceFromClipper(
LPDIRECTDRAWCLIPPER *lpDDClipper*,
LPGUID *lpGUID*,
int *width*,
int *height*,
LPDIRECT3DRMDEVICE2 * *lplpD3DRMDevice*
);

Parameters

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 IDirect3DRMDevice2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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:

D3DPSHADECAPS_ALPHAFLATSTIPPLED
D3DPTBLENDCAPS_COPY | **D3DPTBLENDCAPS_MODULATE**
D3DPMISCCAPS_CULLCCW
D3DPRASTERCAPS_FOGVERTEX
D3DPCMPCAPS_LESSEQUAL
D3DPTFILTERCAPS_NEAREST
D3DPTEXTURECAPS_PERSPECTIVE | **D3DPTEXTURECAPS_TRANSPARENCY**
D3DPTADDRESSCAPS_WRAP

If a hardware device is not found, the monochromatic (ramp) software driver is loaded. An application should enumerate devices instead of specifying NULL in the *lpGUID* parameter if it has special needs that are not met by this list of default settings.

IDirect3DRM2::CreateDeviceFromD3D

Creates a Direct3DRMDevice2 Windows device by using specified Direct3D objects. An **IDirect3DRMDevice2** interface works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

HRESULT CreateDeviceFromD3D(
LPDIRECT3D2 *lpD3D*,
LPDIRECT3DDEVICE2 *lpD3DDev*,
LPDIRECT3DRMDEVICE2 * *lplpD3DRMDevice*
);

Parameters

lpD3D
Address of an instance of Direct3D2.

lpD3DDev
Address of a Direct3D2 device object.

lplpD3DRMDevice
Address that will be filled with a pointer to an IDirect3DRMDevice2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateDeviceFromSurface

Creates a Direct3DRMDevice2 Windows device for rendering from the specified DirectDraw surfaces. An **IDirect3DRMDevice2** interface works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

HRESULT CreateDeviceFromSurface(
LPGUID *lpGUID*,
LPDIRECTDRAW *lpDD*,
LPDIRECTDRAWSURFACE *lpDDSBack*,
LPDIRECT3DRMDEVICE2 * *lplpD3DRMDevice*
);

Parameters

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 IDirect3DRMDevice2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateFace

Creates an instance of the IDirect3DRMFace interface.

HRESULT CreateFace(
LPDIRECT3DRMFACE * *lplpd3drmFace*
);

Parameters

lplpd3drmFace
Address that will be filled with a pointer to an IDirect3DRMFace interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateFrame

Creates a new child frame (a Direct3DRMFrame2 object) of the given parent frame.

HRESULT CreateFrame(
LPDIRECT3DRMFRAME *lpD3DRMFrame*,
LPDIRECT3DRMFRAME2* *lplpD3DRMFrame2*
);

Parameters

lpD3DRMFrame
Address of a DIRECT3DRMFRAME object that is to be the parent of the new frame.

lplpD3DRMFrame
Address that will be filled with a pointer to an IDirect3DRMFrame2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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 associate it with a parent frame later by using the IDirect3DRMFrame2::AddChild method.

See Also

IDirect3DRMFrame2::AddChild

IDirect3DRM2::CreateLight

Creates a new light source with the given type and color.

HRESULT CreateLight(
D3DRMLIGHTTYPE *d3drmltLightType*,
D3DCOLOR *cColor*,
LPDIRECT3DRMLIGHT* *lplpD3DRMLight*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateLightRGB

Creates a new light source with the given type and color.

HRESULT CreateLightRGB(
D3DRMLIGHTTYPE *ltLightType*,
D3DVALUE *vRed*,
D3DVALUE *vGreen*,
D3DVALUE *vBlue*,
LPDIRECT3DRMLIGHT* *lplpD3DRMLight*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateMaterial

Creates a material with the given specular property.

HRESULT CreateMaterial(
    D3DVALUE *vPower*,
    LPDIRECT3DRMMATERIAL * *lplpD3DRMMaterial*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateMesh

Creates a new mesh object with no faces. The mesh is not visible until it is added to a frame.

HRESULT CreateMesh(
    LPDIRECT3DRMMESH* *lplpD3DRMMesh*
    );

Parameters

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateMeshBuilder

Creates a new mesh builder object.

HRESULT CreateMeshBuilder(
    LPDIRECT3DRMMESHBUILDER2* *lplpD3DRMMeshBuilder2*
    );

Parameters

lplpD3DRMMeshBuilder
Address that will be filled with a pointer to an IDirect3DRMMeshBuilder2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateObject

Creates a new object without initializing the object.

HRESULT CreateObject(
REFCLSID *rclsid*,
LPUNKNOWN *pUnkOuter*,
REFIID *riid*,
LPVOID FAR* *ppv*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

An application that calls this method must initialize the object that has been created. (The other creation methods of the IDirect3DRM2 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.

IDirect3DRM2::CreateProgressiveMesh

Creates a new progressive mesh object with no faces. The mesh is not visible until it is added to a frame.

HRESULT CreateProgressiveMesh(
LPDIRECT3DRMPROGRESSIVEMESH* *lplpD3DRMProgressiveMesh*
);

Parameters

lplpD3DRMProgressiveMesh
Address that will be filled with a pointer to an IDirect3DRMProgressiveMesh interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateShadow

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.

HRESULT CreateShadow(
LPDIRECT3DRMVISUAL *lpVisual*,
LPDIRECT3DRMLIGHT *lpLight*,
D3DVALUE *px*,
D3DVALUE *py*,
D3DVALUE *pz*,
D3DVALUE *nx*,
D3DVALUE *ny*,
D3DVALUE *nz*,
LPDIRECT3DRMVISUAL * *lplpShadow*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateTexture

Creates a texture from an image in memory.

HRESULT CreateTexture(
LPD3DRMIMAGE *lpImage*,
LPDIRECT3DRMTEXTURE2* *lplpD3DRMTexture2*
);

Parameters

lpImage
Address of a D3DRMIMAGE structure describing the source for the texture.

lplpD3DRMTexture2
Address that will be filled with a pointer to an IDirect3DRMTexture2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

IDirect3DRM2::CreateTextureFromSurface

Creates a texture from a specified DirectDraw surface.

HRESULT CreateTextureFromSurface(
    LPDIRECTDRAWSURFACE *lpDDS*,
    LPDIRECT3DRMTEXTURE2 * *lplpD3DRMTexture2*
    );

Parameters

lpDDS
Address of the DirectDrawSurface object containing the texture.

lplpD3DRMTexture
Address that will be filled with a pointer to an IDirect3DRMTexture2 interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateUserVisual

Creates an application-defined visual object, which can then be added to a scene and rendered by using an application-defined handler.

HRESULT CreateUserVisual(
D3DRMUSERVISUALCALLBACK *fn*,
LPVOID *lpArg*,
LPDIRECT3DRMUSERVISUAL * *lplpD3DRMUV*
);

Parameters

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.

Return Values

Should return D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::CreateViewport

Creates a viewport on a Direct3DRMDevice2 device with device coordinates (*dwXPos*, *dwYPos*) to (*dwXPos* + *dwWidth*, *dwYPos* + *dwHeight*).

HRESULT CreateViewport(
LPDIRECT3DRMDEVICE2 *lpDev*,
LPDIRECT3DRMFRAME *lpCamera*,
DWORD *dwXPos*,
DWORD *dwYPos*,
DWORD *dwWidth*,
DWORD *dwHeight*,
LPDIRECT3DRMVIEWPORT* *lplpD3DRMViewport*
);

Parameters

lpDev
Address of a Direct3DRMDevice2 device on which the viewport is to be created.

lpCamera
Address of a frame that describes the position and direction of the view.

dwXPos, *dwYPos*, *dwWidth*, and *dwHeight*
Position and size of the viewport, in device coordinates. The viewport size cannot be larger than the physical device or the method will fail.

lplpD3DRMViewport
Address that will be filled with a pointer to an IDirect3DRMViewport interface if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. If the viewport size is larger than the physical device, returns D3DRMERR_BADVALUE. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The viewport displays objects in the scene that contains the camera, with the view direction and up vector taken from the camera. The viewport size cannot be larger than the physical device.

An **IDirect3DRMDevice2** interface works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRM2::CreateWrap

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.

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*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMWrap

IDirect3DRM2::EnumerateObjects

Calls the callback function specified by the *func* parameter on each of the active Direct3DRM objects.

HRESULT EnumerateObjects(
D3DRMOBJECTCALLBACK *func*,
LPVOID *lpArg*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::GetDevices

Returns all the Direct3DRM devices that have been created in the system.

HRESULT GetDevices(
    LPDIRECT3DRMDEVICEARRAY* *lplpDevArray*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::GetNamedObject

Finds a Direct3DRMObject, given its name.

HRESULT GetNamedObject(
const char * *lpName*,
LPDIRECT3DRMOBJECT* *lplpD3DRMObject*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

If the system does not find an object with the name specified in the *lpName* parameter, this method returns D3DRM_OK but the *lplpD3DRMObject* parameter is NULL.

IDirect3DRM2::GetSearchPath

Returns the current search path. For Windows, the path is a list of directories separated by semicolons (;).

HRESULT GetSearchPath(
DWORD * *lpdwSize***,**
LPSTR *lpszPath*
);

Parameters

lpdwSize

Pointer to a DWORD. Should contain the length of *lpszPath* when **GetSearchPath** is called. On return, contains the length of the string in *lpszPath* containing the current search path. This parameter cannot be NULL.

lpszPath

On return, contains a pointer to a null-terminated string specifying the search path. If *lpszPath* equals NULL when **GetSearchPath** is called, the method returns the length of the current string in the location pointed to by the *lpdwSize* parameter.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRM2::SetSearchPath

IDirect3DRM2::Load

Loads an object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
LPIID * *lplpGUIDs*,
DWORD *dwcGUIDs*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADCALLBACK *d3drmLoadProc*,
LPVOID *lpArgLP*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpArgLTP*,
LPDIRECT3DRMFRAME *lpParentFrame*
);

Parameters

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_IDirect3DRMMeshBuilder and IID_IDirect3DRMAnimationSet, this method loads all the animation sets and mesh-builder objects. Possible GUIDs must be one or more of the following: IID_IDirect3DRMMeshBuilder, IID_IDirect3DRMAnimationSet, IID_IDirect3DRMAnimation, and IID_IDirect3DRMFrame.

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 that require special formatting. This parameter can be NULL.

lpArgLTP
Address of application-defined data passed to the D3DRMLOADTEXTURECALLBACK callback function.

lpParentFrame
Address of a parent Direct3DRMFrame object. This argument only affects the loading of animation sets. When an animation that is loaded from an X file references an unparented frame in the X file, its parent is set to this parent frame argument. However, if you ask **Load** to load any frames in the X file, the parent frame argument will not be used as the parent frame for frames in the X file with no parent. That is, the parent frame argument is used only when you load animation sets. This value of this argument can be NULL.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method allows great flexibility in loading objects from DirectX files. See IDirect3DRM::Load for an example of how to use it.

IDirect3DRM2::LoadTexture

Loads a Direct3DRMTexture2 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.

HRESULT LoadTexture(
const char * *lpFileName*,
LPDIRECT3DRMTEXTURE2* *lplpD3DRMTexture*
);

Parameters

lpFileName
Address of the name of the required .bmp or .ppm file.

lplpD3DRMTexture
Address of a pointer to be initialized with a valid IDirect3DRMTexture2 pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

LoadTexture checks whether the texture is in BMP or PPM format, which are the formats it knows how to load. If you want to load other formats, you can add code to the callback to load the image into a D3DRMIMAGE structure and then call IDirect3DRM2::CreateTexture.

If you write your own texture callback, the **LoadTexture** call in the texture callback does not take a reference to the texture. For example:

```
HRESULT loadTextures(char *name, void *arg, LPDIRECT3DRMTEXTURE2 *tex)
{
    return lpD3DRM2->LoadTexture(name, tex);
}
```

In this sample, no reference is taken for *tex*. If you want to keep a reference to each texture loaded by your texture callback, you should **AddRef** the texture. For example:

```
LPDIRECT3DRMTEXTURE2 *texarray;

HRESULT loadTextures(char *name, void *arg, LPDIRECT3DRMTEXTURE2 *tex)
{
  if (FAILED(lpD3DRM2->LoadTexture(name, tex)) {
    return NULL;
  }

  texArray[current++] = tex;
  tex->AddRef();

  return tex;
}
```

IDirect3DRM2::LoadTextureFromResource

Loads a Direct3DRMTexture2 texture from a specified resource.

HRESULT LoadTextureFromResource(
HMODULE *hModule*,
LPCTSTR *strName*,
LPCTSTR *strType*,
LPDIRECT3DRMTEXTURE2 * *lplpD3DRMTexture*
);

Parameters

hModule
Handle of the module whose executable file contains the resource.

strName
A null-terminated string specifying the name of the resource to be used as the texture.

strType
A null-terminated string specifying the type name of the resource. This parameter can be one of the following resource types or a user-defined type:

Value	Meaning
RT_BITMAP	Bitmap reource
RT_RCDATA	Application-defined resource (raw data)

lplpD3DRMTexture
Address of a pointer to be initialized with a valid IDirect3DRMTexture2 object if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRM2::SetDefaultTextureColors

Sets the default colors to be used for a Direct3DRMTexture2 object.

HRESULT SetDefaultTextureColors(
    DWORD *dwColors*
    );

Parameters

dwColors
Number of colors.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method affects the texture colors only when it is called before the IDirect3DRM2::CreateTexture method; it has no effect on textures that have already been created.

IDirect3DRM2::SetDefaultTextureShades

Sets the default shades to be used for an Direct3DRMTexture2 object.

HRESULT SetDefaultTextureShades(
    DWORD *dwShades*
    );

Parameters

dwShades
Number of shades.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method affects the texture shades only when it is called before the IDirect3DRM2::CreateTexture method; it has no effect on textures that have already been created.

IDirect3DRM2::SetSearchPath

Sets the current file search path from a list of directories. For Windows, the path should be a list of directories separated by semicolons (;).

HRESULT SetSearchPath(
    LPCSTR *lpPath*
    );

Parameters

lpPath
Address of a null-terminated string specifying the path to set as the current search path.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

D3DPATH is an environment variable that sets the default search path.

See Also

IDirect3DRM2::GetSearchPath

IDirect3DRM2::Tick

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.

HRESULT Tick(
D3DVALUE *d3dvalTick*
);

Parameters

d3dvalTick

Velocity and rotation step for the IDirect3DRMFrame::SetRotation and IDirect3DRMFrame::SetVelocity methods.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You can implement this method by using other Retained-Mode methods to allow more flexibility in rendering a scene.

IDirect3DRMAnimation

Applications use the methods of the **IDirect3DRMAnimation** interface to animate the position, orientation, and scaling of visuals, lights, and viewports. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMAnimation and IDirect3DRMAnimationSet Interfaces.

The methods of the **IDirect3DRMAnimation** interface can be organized into the following groups:

Keys	AddPositionKey
	AddRotateKey
	AddScaleKey
	DeleteKey
Miscellaneous	SetFrame
	SetTime
Options	GetOptions
	SetOptions

The **IDirect3DRMAnimation** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Adds a position key to the animation.

HRESULT AddPositionKey(
D3DVALUE *rvTime*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The transformation applied by this method is a translation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

See Also

IDirect3DRMAnimation::DeleteKey

IDirect3DRMAnimation::AddRotateKey

Adds a rotate key to the animation.

HRESULT AddRotateKey(
D3DVALUE *rvTime*,
D3DRMQUATERNION **rqQuat*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method applies a rotation transformation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

See Also

IDirect3DRMAnimation::DeleteKey

IDirect3DRMAnimation::AddScaleKey

Adds a scale key to the animation.

HRESULT AddScaleKey(
D3DVALUE *rvTime*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method applies a scaling transformation. For information about the matrix mathematics involved in transformations, see 3D Transformations.

See Also

IDirect3DRMAnimation::DeleteKey

IDirect3DRMAnimation::DeleteKey

Removes a key from an animation.

HRESULT DeleteKey(
D3DVALUE *rvTime*
);

Parameters

rvTime
Time identifying the key that will be removed from the animation.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMAnimation::GetOptions

Retrieves animation options.

D3DRMANIMATIONOPTIONS GetOptions( );

Return Values

Returns the value of the D3DRMANIMATIONOPTIONS type describing the animation options.

See Also

IDirect3DRMAnimation::SetOptions

IDirect3DRMAnimation::SetFrame

Sets the frame for the animation.

HRESULT SetFrame(
    LPDIRECT3DRMFRAME *lpD3DRMFrame*
    );

Parameters

lpD3DRMFrame
Address of a variable representing the frame to set for the animation.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMAnimation::SetOptions

Sets the animation options.

HRESULT SetOptions(
D3DRMANIMATIONOPTIONS *d3drmanimFlags*
);

Parameters

d3drmanimFlags
Value of the D3DRMANIMATIONOPTIONS type describing the animation options.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMAnimation::GetOptions

IDirect3DRMAnimation::SetTime

Sets the current time for this animation.

HRESULT SetTime(
    D3DVALUE *rvTime*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMAnimationSet

Applications use the methods of the **IDirect3DRMAnimationSet** interface to group Direct3DRMAnimation objects together, which can simplify the playback of complex animation sequences. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMAnimation and IDirect3DRMAnimationSet Interfaces.

The methods of the **IDirect3DRMAnimationSet** interface can be organized into the following groups:

Adding, loading, and removing	AddAnimation DeleteAnimation Load
Time	SetTime

The **IDirect3DRMAnimationSet** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Adds an animation to the animation set.

HRESULT AddAnimation(
    LPDIRECT3DRMANIMATION *lpD3DRMAnimation*
    );

Parameters

lpD3DRMAnimation
Address of the Direct3DRMAnimation object to be added to the animation set.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMAnimationSet::DeleteAnimation

Removes a previously added animation from the animation set.

HRESULT DeleteAnimation(
    LPDIRECT3DRMANIMATION *lpD3DRMAnimation*
    );

Parameters

lpD3DRMAnimation
Address of the Direct3DRMAnimation object to be removed from the animation set.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMAnimationSet::Load

Loads an animation set.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpArgLTP*,
LPDIRECT3DRMFRAME *lpParentFrame*
);

Parameters

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 that require special formatting. This parameter can be NULL.

lpArgLTP
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

lpParentFrame
Address of a parent Direct3DRMFrame object. This argument only affects the loading of animation sets. When an animation that is loaded from an X file references an unparented frame in the X file, its parent is set to this parent frame argument. However, if you ask **Load** to load any frames in the X file, the parent frame argument will not be used as the parent frame for frames in the X file with no parent. That is, the parent frame argument is used only when you load animation sets. This value of this argument can be NULL.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

By default, this method loads the first animation set in the file specified by the *lpvObjSource* parameter.

IDirect3DRMAnimationSet::SetTime

Sets the time for this animation set.

HRESULT SetTime(
    D3DVALUE *rvTime*
    );

Parameters

rvTime
New time.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice

Applications use the methods of the **IDirect3DRMDevice** interface to interact with the output device. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMDevice, IDirect3DRMDevice2, and IDirect3DRMDeviceArray Interfaces.

The methods of the **IDirect3DRMDevice** interface can be organized into the following groups:

Buffer counts	GetBufferCount SetBufferCount
Color models	GetColorModel
Dithering	GetDither SetDither
Initialization	Init InitFromClipper InitFromD3D
Miscellaneous	GetDirect3DDevice GetHeight GetTrianglesDrawn GetViewports GetWidth GetWireframeOptions Update
Notifications	AddUpdateCallback DeleteUpdateCallback
Rendering quality	GetQuality SetQuality
Shading	GetShades SetShades
Texture quality	GetTextureQuality SetTextureQuality

The **IDirect3DRMDevice** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::AddUpdateCallback

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.

HRESULT AddUpdateCallback(
D3DRMUPDATECALLBACK *d3drmUpdateProc*,
LPVOID *arg*
);

Parameters

d3drmUpdateProc
Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg
Private data to be passed to the update callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice::DeleteUpdateCallback, IDirect3DRMDevice::Update, D3DRMUPDATECALLBACK

IDirect3DRMDevice::DeleteUpdateCallback

Removes an update callback function that was added by calling the IDirect3DRMDevice::AddUpdateCallback method.

HRESULT DeleteUpdateCallback(
D3DRMUPDATECALLBACK *d3drmUpdateProc*,
LPVOID *arg*
);

Parameters

d3drmUpdateProc
Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg
Private data that was passed to the update callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice::AddUpdateCallback, IDirect3DRMDevice::Update, D3DRMUPDATECALLBACK

IDirect3DRMDevice::GetBufferCount

Retrieves the value set in a call to the IDirect3DRMDevice::SetBufferCount method.

DWORD GetBufferCount( );

Return Values

Returns the number of buffers—one for single-buffering, two for double-buffering, and so on.

IDirect3DRMDevice::GetColorModel

Retrieves the color model of a device.

D3DCOLORMODEL GetColorModel( );

Return Values

Returns a value from the **D3DCOLORMODEL** enumerated type that describes the Direct3D color model (RGB or monochrome).

See Also

Color Models

IDirect3DRMDevice::GetDirect3DDevice

Retrieves a pointer to an Immediate-Mode device.

HRESULT GetDirect3DDevice(
    LPDIRECT3DDEVICE * *lplpD3DDevice*
    );

Parameters

lplpD3DDevice
    Address of a pointer that is initialized with a pointer to an Immediate-Mode device object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice::GetDither

Retrieves the dither flag for the device.

BOOL GetDither( );

Return Values

Returns TRUE if the dither flag is set, or FALSE otherwise.

See Also

IDirect3DRMDevice::SetDither

IDirect3DRMDevice::GetHeight

Retrieves the height, in pixels, of a device. This method is a convenience function.

DWORD GetHeight( );

Return Values

Returns the height.

IDirect3DRMDevice::GetQuality

Retrieves the rendering quality for the device.

D3DRMRENDERQUALITY GetQuality( );

Return Values

Returns one or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type.

See Also

IDirect3DRMDevice::SetQuality

IDirect3DRMDevice::GetShades

Retrieves the number of shades in a ramp of colors used for shading.

DWORD GetShades( );

Return Values

Returns the number of shades.

See Also

IDirect3DRMDevice::SetShades

IDirect3DRMDevice::GetTextureQuality

Retrieves the current texture quality parameter for the device. Texture quality is relevant only for an RGB device.

D3DRMTEXTUREQUALITY GetTextureQuality( );

Return Values

Returns one of the members of the D3DRMTEXTUREQUALITY enumerated type.

See Also

IDirect3DRMDevice::SetTextureQuality

IDirect3DRMDevice::GetTrianglesDrawn

Retrieves the number of triangles drawn to a device since its creation. This method is a convenience function.

DWORD GetTrianglesDrawn( );

Return Values

Returns the number of triangles.

Remarks

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::GetViewports

Constructs a Direct3DRMViewportArray object that represents the viewports currently constructed from the device.

HRESULT GetViewports(
LPDIRECT3DRMVIEWPORTARRAY* *lplpViewports*
);

Parameters

lplpViewports
Address of a pointer that is initialized with a valid Direct3DRMViewportArray object if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice::GetWidth

Retrieves the width, in pixels, of a device. This method is a convenience function.

DWORD GetWidth( );

Return Values

Returns the width.

IDirect3DRMDevice::GetWireframeOptions

Retrieves the wireframe options of a given device.

DWORD GetWireframeOptions( );

Return Values

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

Not implemented on the Windows platform.

HRESULT Init(
    ULONG *width*,
    ULONG *height*
    );

IDirect3DRMDevice::InitFromClipper

Initializes a device from a specified DirectDrawClipper object.

HRESULT InitFromClipper(
LPDIRECTDRAWCLIPPER *lpDDClipper*,
LPGUID *lpGUID*,
int *width*,
int *height*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice::InitFromD3D

Initializes a Retained-Mode device from a specified Direct3D Immediate-Mode object and Immediate-Mode device.

HRESULT InitFromD3D(
LPDIRECT3D *lpD3D*,
LPDIRECT3DDEVICE *lpD3DIMDev*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice::SetBufferCount

Sets the number of buffers currently being used by the application.

HRESULT SetBufferCount(
    DWORD *dwCount*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMDevice::GetBufferCount

IDirect3DRMDevice::SetDither

Sets the dither flag for the device.

HRESULT SetDither(
BOOL *bDither*
);

Parameters

bDither
New dithering mode for the device. The default is TRUE.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice::GetDither

IDirect3DRMDevice::SetQuality

Sets the rendering quality of a device

HRESULT SetQuality (
D3DRMRENDERQUALITY *rqQuality*
);

Parameters

rqQuality

One or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type. The default setting is D3DRMRENDER_FLAT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The rendering quality is the maximum quality at which rendering can take place on the rendering surface of that device.

An object's quality has three components: shade mode (flat or gouraud, phong is not yet implemented and will default to gouraud shading), lighting type (on or off), and fill mode (point, wireframe or solid).

You set the quality of a device with **SetQuality**. By default it is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMProgressiveMesh, Direct3DRMMeshBuilder, or Direct3DRMMeshBuilder2 object with their respective **SetQuality** methods. By default, the quality of these objects is D3DRMRENDER_GOURAUD (gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

See Also

IDirect3DRMDevice::GetQuality

IDirect3DRMDevice::SetShades

Sets the number of shades in a ramp of colors used for shading.

HRESULT SetShades(
    DWORD *ulShades*
    );

Parameters

ulShades
New number of shades. This parameter must be a power of 2. The default is 32.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice::GetShades

IDirect3DRMDevice::SetTextureQuality

Sets the texture quality for the device.

HRESULT SetTextureQuality(
    D3DRMTEXTUREQUALITY *tqTextureQuality*
    );

Parameters

tqTextureQuality
One of the members of the D3DRMTEXTUREQUALITY enumerated type. The default is D3DRMTEXTURE_NEAREST.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice::GetTextureQuality

IDirect3DRMDevice::Update

Copies the image that has been rendered to the display. It also provides a heartbeat function to the device driver.

HRESULT Update( );

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMDevice::AddUpdateCallback, D3DRMUPDATECALLBACK

IDirect3DRMDevice2

Applications use the methods of the IDirect3DRMDevice2 and IDirect3DRMDevice interfaces to interact with the output device. While IDirect3DRMDevice, when created from the IDirect3DRM interface, works with an **IDirect3DDevice** Immediate-Mode device, **IDirect3DRMDevice2**, when created from the IDirect3DRM2 interface, or initialized by the IDirect3DRMDevice2::InitFromD3D2 or IDirect3DRMDevice2::InitFromSurface method, works with an **IDirect3DDevice2** Immediate-Mode device. The **IDirect3DDevice2** device supports the **DrawPrimitive** interface as well as execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

The IDirect3DRMDevice2::InitFromSurface method uses the IDirect3DRM2::CreateDevice method to create an IDirect3DRMDevice2 interface. The IDirect3DRMDevice2::InitFromD3D2 method uses an **IDirect3D2** Immediate-Mode object and an **IDirect3DDevice2** Immediate-Mode device to initialize an **IDirect3DRMDevice2** Retained-Mode device.

You can still query back and forth between IDirect3DRMDevice and **IDirect3DRMDevice2**. The main difference is in how the underlying Immediate-Mode device is created.

IDirect3DRMDevice2 contains all the methods in IDirect3DRMDevice plus two addition ones that allow you to control transparency, IDirect3DRMDevice2::GetRenderMode and IDirect3DRMDevice2::SetRenderMode, one additional initialization method, IDirect3DRMDevice2::InitFromSurface, and two changed methods IDirect3DRMDevice2::GetDirect3DDevice2 and IDirect3DRMDevice2::InitFromD3D2 which get and initialize an **IDirect3DRMDevice2** object rather than an **IDirect3DRMDevice** object.

This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMDevice, IDirect3DRMDevice2, and IDirect3DRMDeviceArray Interfaces.

The methods of the **IDirect3DRMDevice2** interface can be organized into the following groups:

Buffer counts	GetBufferCount SetBufferCount
Color models	GetColorModel
Dithering	GetDither SetDither
Initialization	Init InitFromClipper InitFromD3D2 InitFromSurface
Miscellaneous	GetDirect3DDevice2 GetHeight GetTrianglesDrawn GetViewports GetWidth GetWireframeOptions Update
Notifications	AddUpdateCallback DeleteUpdateCallback
Rendering quality	GetQuality SetQuality

Shading	GetShades SetShades
Texture quality	GetTextureQuality SetTextureQuality
Transparency	GetRenderMode SetRenderMode

The **IDirect3DRMDevice2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The **IUnknown** interface supports the following three methods:

AddRef
QueryInterface
Release

In addition, the **IDirect3DRMDevice2** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMDevice2 object is obtained by calling the IDirect3DRM2::CreateDevice method.

IDirect3DRMDevice2::AddUpdateCallback

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 IDirect3DRMDevice2::Update method.

HRESULT AddUpdateCallback(
D3DRMUPDATECALLBACK *d3drmUpdateProc*,
LPVOID *arg*
);

Parameters

d3drmUpdateProc
Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg
Private data to be passed to the update callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice2::DeleteUpdateCallback, IDirect3DRMDevice2::Update, D3DRMUPDATECALLBACK

IDirect3DRMDevice2::DeleteUpdateCallback

Removes an update callback function that was added by calling the IDirect3DRMDevice2::AddUpdateCallback method.

HRESULT DeleteUpdateCallback(
D3DRMUPDATECALLBACK *d3drmUpdateProc*,
LPVOID *arg*
);

Parameters

d3drmUpdateProc
Address of an application-defined callback function, D3DRMUPDATECALLBACK.

arg
Private data that was passed to the update callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice2::AddUpdateCallback, IDirect3DRMDevice2::Update, D3DRMUPDATECALLBACK

IDirect3DRMDevice2::GetBufferCount

Retrieves the value set in a call to the IDirect3DRMDevice2::SetBufferCount method.

DWORD GetBufferCount( );

Return Values

Returns the number of buffers—one for single-buffering, two for double-buffering, and so on.

IDirect3DRMDevice2::GetColorModel

Retrieves the color model of a device.

D3DCOLORMODEL GetColorModel( );

Return Values

Returns a value from the **D3DCOLORMODEL** enumerated type that describes the Direct3D color model (RGB or monochrome).

See Also

Color Models

IDirect3DRMDevice2::GetDirect3DDevice2

Retrieves a pointer to an **IDirect3DDevice2** Immediate-Mode device.

HRESULT GetDirect3DDevice2(
LPDIRECT3DDEVICE2 * *lplpD3DDevice*
);

Parameters

lplpD3DDevice
Address of a pointer that is initialized with a pointer to an **IDirect3DDevice2** Immediate-Mode device object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The **IDirect3DDevice2** device will support the DrawPrimitive interface and execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRMDevice2::GetDither

Retrieves the dither flag for the device.

BOOL GetDither( );

Return Values

Returns TRUE if the dither flag is set, or FALSE otherwise.

See Also

IDirect3DRMDevice2::SetDither

IDirect3DRMDevice2::GetHeight

Retrieves the height, in pixels, of a device. This method is a convenience function.

DWORD GetHeight( );

Return Values

Returns the height.

IDirect3DRMDevice2::GetQuality

Retrieves the rendering quality for the device.

D3DRMRENDERQUALITY GetQuality( );

Return Values

Returns one or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type.

See Also

IDirect3DRMDevice2::SetQuality

IDirect3DRMDevice2::GetRenderMode

Retrieves the current transparency flags.

DWORD GetRenderMode( );

Return Values

Returns the value of the current transparency flags.

Remarks

Transparency flags have the following values:

Flag	Value
No flag set (default)	0
D3DRMRENDERMODE_BLENDEDTRANSPARENCY	1
D3DRMRENDERMODE_SORTEDTRANSPARENCY	2

See Also

IDirect3DRMDevice2::SetRenderMode

IDirect3DRMDevice2::GetShades

Retrieves the number of shades in a ramp of colors used for shading.

DWORD GetShades( );

Return Values

Returns the number of shades.

See Also

IDirect3DRMDevice2::SetShades

IDirect3DRMDevice2::GetTextureQuality

Retrieves the current texture quality parameter for the device. Texture quality is relevant only for an RGB device.

D3DRMTEXTUREQUALITY GetTextureQuality( );

Return Values

Returns one of the members of the D3DRMTEXTUREQUALITY enumerated type.

See Also

IDirect3DRMDevice2::SetTextureQuality

IDirect3DRMDevice2::GetTrianglesDrawn

Retrieves the number of triangles drawn to a device since its creation. This method is a convenience function.

DWORD GetTrianglesDrawn( );

Return Values

Returns the number of triangles.

Remarks

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.

IDirect3DRMDevice2::GetViewports

Constructs a Direct3DRMViewportArray object that represents the viewports currently constructed from the device.

HRESULT GetViewports(
LPDIRECT3DRMVIEWPORTARRAY* *lplpViewports*
);

Parameters

lplpViewports
Address of a pointer that is initialized with a valid Direct3DRMViewportArray object if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice2::GetWidth

Retrieves the width, in pixels, of a device. This method is a convenience function.

DWORD GetWidth( );

Return Values

Returns the width.

IDirect3DRMDevice2::GetWireframeOptions

Retrieves the wireframe options of a given device.

DWORD GetWireframeOptions( );

Return Values

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.

IDirect3DRMDevice2::Init

Not implemented on the Windows platform.

HRESULT Init(
ULONG *width*,
ULONG *height*
);

IDirect3DRMDevice2::InitFromClipper

Initializes an **IDirect3DDevice2** device from a specified DirectDrawClipper object using IDirect3DRM2::CreateDevice.

HRESULT InitFromClipper(
LPDIRECTDRAWCLIPPER *lpDDClipper*,
LPGUID *lpGUID*,
int *width*,
int *height*
);

Parameters

lpDDClipper
Address of the DirectDrawClipper object to use as an initializer.
lpGUID the Direct3D device driver to use.
width and *height*
Width and height of the device.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMDevice2::InitFromD3D2

Initializes an **IDirect3DRMDevice2** Retained-Mode device from an **IDirect3D2** Immediate-Mode object and an **IDirect3DDevice2** Immediate-Mode device.

HRESULT InitFromD3D2(
LPDIRECT3D2 *lpD3D*,
LPDIRECT3DDEVICE2 *lpD3DIMDev*
);

Parameters

lpD3D

Address of the **IDirect3D2** Immediate-Mode object to use to initialize the Retained-Mode device.

lpD3DIMDev

Address of the **IDirect3DDevice2** Immediate-Mode device to use to initialize the Retained-Mode device.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The **IDirect3DRMDevice2** device initialized from **IDirect3DDevice2** will support the DrawPrimitive interface and execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRMDevice2::InitFromSurface

Initializes a **IDirect3DDevice2** device from a specified DirectDraw surface, using the IDirect3DRM2::CreateDevice method.

HRESULT InitFromSurface(
LPGUID *lpGUID*,
LPDIRECTDRAW *lpDD*,
LPDIRECTDRAWSURFACE *lpDDSBack*
);

Parameters

lpGUID
Address of the globally unique identifier (GUID) that identifies the Direct3D device driver to use.

lpDD
Address of the interface of a DirectDraw object that created the DirectDrawSurface.

lpDDSBack
Address of the interface of a DirectDrawSurface back buffer onto which the device will render.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The **IDirect3DRMDevice2** device initialized will support the DrawPrimitive interface and execute buffers, and is required for progressive meshes and for alpha blending and sorting of transparent objects.

IDirect3DRMDevice2::SetBufferCount

Sets the number of buffers currently being used by the application.

HRESULT SetBufferCount(
    DWORD *dwCount*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMDevice2::GetBufferCount

IDirect3DRMDevice2::SetDither

Sets the dither flag for the device.

HRESULT SetDither(
BOOL *bDither*
);

Parameters

bDither
New dithering mode for the device. The default is TRUE.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice2::GetDither

IDirect3DRMDevice2::SetQuality

Sets the rendering quality of a device

HRESULT SetQuality (
D3DRMRENDERQUALITY *rqQuality*
);

Parameters

rqQuality

One or more of the members of the enumerated types represented by the D3DRMRENDERQUALITY type. The default setting is D3DRMRENDER_FLAT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The rendering quality is the maximum quality at which rendering can take place on the rendering surface of that device.

An object's quality has three components: shade mode (flat or gouraud, phong is not yet implemented and will default to gouraud shading), lighting type (on or off), and fill mode (point, wireframe or solid).

You set the quality of a device with **SetQuality**. By default it is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMProgressiveMesh, Direct3DRMMeshBuilder, or Direct3DRMMeshBuilder2 object with their respective **SetQuality** methods. By default, the quality of these objects is D3DRMRENDER_GOURAUD (gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

See Also

IDirect3DRMDevice2::GetQuality

IDirect3DRMDevice2::SetRenderMode

Sets the transparency mode. The mode type determines how transparent objects will be rendered. The default mode renders transparent objects with stippled transparency.

HRESULT SetRenderMode(
DWORD *dwFlags*
);

Parameters

dwFlags

One or more of the transparent mode flags. The default (*dwFlags* = 0) sets the transparency mode to stippled transparency. In addition, flags can have one or more of the following values:

D3DRMRENDERMODE_BLENDEDTRANSPARENCY (*dwFlags* = 1) sets the transparency mode to alpha blending.

D3DRMRENDERMODE_SORTEDTRANSPARENCY (*dwFlags* = 2) sets the transparency mode so that transparent polygons in the scene are buffered, sorted, and rendered in a second pass. This flag has no effect if the D3DRMRENDERMODE_BLENDEDTRANSPARENCY flag is not also set.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

If the D3DRMRENDERMODE_BLENDEDTRANSPARENCY and D3DRMRENDERMODE_SORTEDTRANSPARENCY flags are set together, it ensures that when two transparent objects are rendered one on top of the other, the image will blend in the correct order to ensure the right visual result.

See Also

IDirect3DRMDevice2::GetRenderMode

IDirect3DRMDevice2::SetShades

Sets the number of shades in a ramp of colors used for shading.

HRESULT SetShades(
    DWORD *ulShades*
    );

Parameters

ulShades
New number of shades. This parameter must be a power of 2. The default is 32.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice2::GetShades

IDirect3DRMDevice2::SetTextureQuality

Sets the texture quality for the device.

HRESULT SetTextureQuality(
D3DRMTEXTUREQUALITY *tqTextureQuality*
);

Parameters

tqTextureQuality

One of the members of the D3DRMTEXTUREQUALITY enumerated type. The default is D3DRMTEXTURE_NEAREST.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMDevice2::GetTextureQuality

IDirect3DRMDevice2::Update

Copies the image that has been rendered to the display. It also provides a heartbeat function to the device driver.

HRESULT Update( );

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Each call to this method causes the system to call an application-defined callback function, D3DRMUPDATECALLBACK. To add a callback function, use the IDirect3DRMDevice2::AddUpdateCallback method.

See Also

IDirect3DRMDevice2::AddUpdateCallback, D3DRMUPDATECALLBACK

IDirect3DRMFace

Applications use the methods of the **IDirect3DRMFace** interface to interact with a single polygon in a mesh. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMFace and IDirect3DRMFaceArray Interfaces.

The methods of the **IDirect3DRMFace** interface can be organized into the following groups:

Color	GetColor SetColor SetColorRGB
Materials	GetMaterial SetMaterial
Textures	GetTexture GetTextureCoordinateIndex GetTextureCoordinates GetTextureTopology SetTexture SetTextureCoordinates SetTextureTopology
Vertices and normals	AddVertex AddVertexAndNormalIndexed GetNormal GetVertex GetVertexCount GetVertexIndex GetVertices

The **IDirect3DRMFace** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::AddVertex

Adds a vertex to a Direct3DRMFace object.

HRESULT AddVertex(
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

x, y, and *z*
The x, y, and z components of the position of the new vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::AddVertexAndNormalIndexed

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.

HRESULT AddVertexAndNormalIndexed(
DWORD *vertex*,
DWORD *normal*
);

Parameters

vertex and *normal*
Indexes of the vertex and normal to add.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::GetColor

Retrieves the color of a Direct3DRMFace object.

D3DCOLOR GetColor( );

Return Values

Returns the color.

See Also

IDirect3DRMFace::SetColor

IDirect3DRMFace::GetMaterial

Retrieves the material of a Direct3DRMFace object.

HRESULT GetMaterial(
    LPDIRECT3DRMMATERIAL* *lplpMaterial*
    );

Parameters

lplpMaterial

Address of a variable that will be filled with a pointer to the Direct3DRMMaterial object applied to the face.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::SetMaterial

IDirect3DRMFace::GetNormal

Retrieves the normal vector of a Direct3DRMFace object.

HRESULT GetNormal(
D3DVECTOR **lpNormal*
);

Parameters

lpNormal
Address of a **D3DVECTOR** structure that will be filled with the normal vector of the face.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::GetTexture

Retrieves the Direct3DRMTexture object applied to a Direct3DRMFace object.

HRESULT GetTexture(
    LPDIRECT3DRMTEXTURE* *lplpTexture*
    );

Parameters

lplpTexture
Address of a variable that will be filled with a pointer to the texture applied to the face.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::SetTexture

IDirect3DRMFace::GetTextureCoordinateIndex

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.

int GetTextureCoordinateIndex(
    DWORD *dwIndex*
    );

Parameters

dwIndex
Index within the face of the vertex.

Return Values

Returns the index.

IDirect3DRMFace::GetTextureCoordinates

Retrieves the texture coordinates of a vertex in a Direct3DRMFace object.

HRESULT GetTextureCoordinates(
DWORD *index*,
D3DVALUE **lpU*,
D3DVALUE **lpV*
);

Parameters

index
Index of the vertex.

lpU and *lpV*
Addresses of variables that are filled with the texture coordinates of the vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::GetTextureTopology

Retrieves the texture topology of a Direct3DRMFace object.

HRESULT GetTextureTopology(
BOOL **lpU*,
BOOL **lpV*
);

Parameters

lpU and *lpV*
Addresses of variables that are set or cleared depending on how the cylindrical wrapping flags are set for the face.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::SetTextureTopology

IDirect3DRMFace::GetVertex

Retrieves the position and normal of a vertex in a Direct3DRMFace object.

HRESULT GetVertex(
DWORD *index*,
D3DVECTOR **lpPosition*,
D3DVECTOR **lpNormal*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::GetVertexCount

Retrieves the number of vertices in a Direct3DRMFace object.

int GetVertexCount( );

Return Values

Returns the number of vertices.

IDirect3DRMFace::GetVertexIndex

Retrieves the index of the vertex in the face's mesh. This index corresponds to the index specified in the *dwIndex* parameter.

int GetVertexIndex (
    DWORD *dwIndex*
    );

Parameters

dwIndex
Index within the face of the vertex.

Return Values

Returns the index.

IDirect3DRMFace::GetVertices

Retrieves the position and normal vector of each vertex in a Direct3DRMFace object.

HRESULT GetVertices(
    DWORD **lpdwVertexCount*,
    D3DVECTOR **lpPosition*,
    D3DVECTOR **lpNormal*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::SetColor

Sets a Direct3DRMFace object to a given color.

HRESULT SetColor(
D3DCOLOR *color*
);

Parameters

color
Color to set.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::GetColor

IDirect3DRMFace::SetColorRGB

Sets a Direct3DRMFace object to a given color.

HRESULT SetColorRGB(
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

red, green, and *blue*
The red, green, and blue components of the color.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::SetMaterial

Sets the material of a Direct3DRMFace object.

HRESULT SetMaterial(
    LPDIRECT3DRMMATERIAL *lpD3DRMMaterial*
    );

Parameters

lpD3DRMMaterial
Address of the material.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::GetMaterial

IDirect3DRMFace::SetTexture

Sets the texture of a Direct3DRMFace object.

HRESULT SetTexture(
    LPDIRECT3DRMTEXTURE *lpD3DRMTexture*
    );

Parameters

lpD3DRMTexture
Address of the texture.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::GetTexture

IDirect3DRMFace::SetTextureCoordinates

Sets the texture coordinates of a specified vertex in a Direct3DRMFace object.

HRESULT SetTextureCoordinates(
DWORD *vertex*,
D3DVALUE *u*,
D3DVALUE *v*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFace::SetTextureTopology

Sets the texture topology of a Direct3DRMFace object.

HRESULT SetTextureTopology(
BOOL *cylU*,
BOOL *cylV*
);

Parameters

cylU and *cylV*
Specify whether the texture has a cylindrical topology in the u and v dimensions.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFace::GetTextureTopology

IDirect3DRMFrame

Applications use the methods of the **IDirect3DRMFrame** interface to interact with frames—an object's frame of reference. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMFrame, IDirect3DRMFrame2, and IDirect3DRMFrameArray Interfaces.

The methods of the **IDirect3DRMFrame** interface 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
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

The **IDirect3DRMFrame** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Adds a child frame to a frame hierarchy.

HRESULT AddChild(
    LPDIRECT3DRMFRAME *lpD3DRMFrameChild*
    );

Parameters

lpD3DRMFrameChild
Address of the Direct3DRMFrame object that will be added as a child.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

To preserve an object's transformation, use the IDirect3DRMFrame::GetTransform method to retrieve the object's transformation before using the **AddChild** method. Then reapply the transformation after the frame is added.

See Also

Hierarchies

IDirect3DRMFrame::AddLight

Adds a light to a frame.

HRESULT AddLight(
    LPDIRECT3DRMLIGHT *lpD3DRMLight*
    );

Parameters

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be added to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::AddMoveCallback

Adds a callback function for special movement processing.

HRESULT AddMoveCallback(
D3DRMFRAMEMOVECALLBACK *d3drmFMC*,
VOID * *lpArg*
);

Parameters

d3drmFMC
Application-defined D3DRMFRAMEMOVECALLBACK callback function.

lpArg
Application-defined data to be passed to the callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::Move, IDirect3DRMFrame::DeleteMoveCallback

IDirect3DRMFrame::AddRotation

Adds a rotation about (*rvX*, *rvY*, *rvZ*) by the number of radians specified in *rvTheta*.

HRESULT AddRotation(
D3DRMCOMBINETYPE *rctCombine***,**
D3DVALUE *rvX***,**
D3DVALUE *rvY***,**
D3DVALUE *rvZ***,**
D3DVALUE *rvTheta*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

3D Transformations, IDirect3DRMFrame::SetRotation

IDirect3DRMFrame::AddScale

Scales a frame's local transformation by (*rvX*, *rvY*, *rvZ*).

HRESULT AddScale(
D3DRMCOMBINETYPE *rctCombine*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified transformation changes the matrix only for the frame identified by this IDirect3DRMFrame interface.

See Also

3D Transformations

IDirect3DRMFrame::AddTransform

Transforms the local coordinates of the frame by the given affine transformation according to the value of the *rctCombine* parameter.

HRESULT AddTransform(
D3DRMCOMBINETYPE *rctCombine*,
D3DRMMATRIX4D *rmMatrix*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Although a 4×4 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.

See Also

3D Transformations

IDirect3DRMFrame::AddTranslation

Adds a translation by (*rvX*, *rvY*, *rvZ*) to a frame's local coordinate system.

HRESULT AddTranslation(
D3DRMCOMBINETYPE *rctCombine*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified translation changes the matrix only for the frame identified by this IDirect3DRMFrame interface.

See Also

3D Transformations

IDirect3DRMFrame::AddVisual

Adds a visual object to a frame.

HRESULT AddVisual(
    LPDIRECT3DRMVISUAL *lpD3DRMVisual*
    );

Parameters

lpD3DRMVisual
Address of a variable that represents the Direct3DRMVisual object to be added to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Removes a frame from the hierarchy. If the frame is not referenced, it is destroyed along with any child frames, lights, and meshes.

HRESULT DeleteChild(
    LPDIRECT3DRMFRAME *lpChild*
    );

Parameters

lpChild
Address of a variable that represents the Direct3DRMFrame object to be used as the child.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

Hierarchies

IDirect3DRMFrame::DeleteLight

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.

HRESULT DeleteLight(
    LPDIRECT3DRMLIGHT *lpD3DRMLight*
    );

Parameters

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be removed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::DeleteMoveCallback

Removes a callback function that performed special movement processing.

HRESULT DeleteMoveCallback(
D3DRMFRAMEMOVECALLBACK *d3drmFMC*,
VOID * *lpArg*
);

Parameters

d3drmFMC
Application-defined D3DRMFRAMEMOVECALLBACK callback function.

lpArg
Application-defined data that was passed to the callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::AddMoveCallback, IDirect3DRMFrame::Move

IDirect3DRMFrame::DeleteVisual

Removes a visual object from a frame, destroying it if it is no longer referenced.

HRESULT DeleteVisual(
    LPDIRECT3DRMVISUAL *lpD3DRMVisual*
    );

Parameters

lpD3DRMVisual
Address of a variable that represents the Direct3DRMVisual object to be removed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::GetChildren

Retrieves a list of child frames in the form of a Direct3DRMFrameArray object.

HRESULT GetChildren(
    LPDIRECT3DRMFRAMEARRAY* *lplpChildren*
    );

Parameters

lplpChildren
Address of a pointer to be initialized with a valid Direct3DRMFrameArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

Direct3DRMFrameArray, Hierarchies

IDirect3DRMFrame::GetColor

Retrieves the color of the frame.

D3DCOLOR GetColor( );

Return Values

Returns the color of the Direct3DRMFrame object.

See Also

IDirect3DRMFrame::SetColor

IDirect3DRMFrame::GetLights

Retrieves a list of lights in the frame in the form of a Direct3DRMLightArray object.

HRESULT GetLights(
    LPDIRECT3DRMLIGHTARRAY* *lplpLights*
    );

Parameters

lplpLights
Address of a pointer to be initialized with a valid Direct3DRMLightArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMLightArray

IDirect3DRMFrame::GetMaterialMode

Retrieves the material mode of the frame.

D3DRMMATERIALMODE GetMaterialMode( );

Return Values

Returns a member of the D3DRMMATERIALMODE enumerated type that specifies the current material mode.

See Also

IDirect3DRMFrame::SetMaterialMode

IDirect3DRMFrame::GetOrientation

Retrieves the orientation of a frame relative to the given reference frame.

HRESULT GetOrientation(
    LPDIRECT3DRMFRAME *lpRef*,
    LPD3DVECTOR *lprvDir*,
    LPD3DVECTOR *lprvUp*
    );

Parameters

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-axis and y-axis, respectively.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetOrientation

IDirect3DRMFrame::GetParent

Retrieves the parent frame of the current frame.

HRESULT GetParent(
    LPDIRECT3DRMFRAME* *lplpParent*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::GetPosition

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.

HRESULT GetPosition(
LPDIRECT3DRMFRAME *lpRef*,
LPD3DVECTOR *lprvPos*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetPosition

IDirect3DRMFrame::GetRotation

Retrieves the rotation of the frame relative to the given reference frame.

HRESULT GetRotation(
    LPDIRECT3DRMFRAME *lpRef*,
    LPD3DVECTOR *lprvAxis*,
    LPD3DVALUE *lprvTheta*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetRotation, Transformations

IDirect3DRMFrame::GetScene

Retrieves the root frame of the hierarchy containing the given frame.

HRESULT GetScene(
    LPDIRECT3DRMFRAME* *lplpRoot*
    );

Parameters

lplpRoot

Address of the pointer that will be filled with the pointer to the Direct3DRMFrame object representing the scene's root frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::GetSceneBackground

Retrieves the background color of a scene.

D3DCOLOR GetSceneBackground( );

Return Values

Returns the color.

IDirect3DRMFrame::GetSceneBackgroundDepth

Retrieves the current background-depth buffer for the scene.

HRESULT GetSceneBackgroundDepth(
    LPDIRECTDRAWSURFACE * *lplpDDSurface*
    );

Parameters

lplpDDSurface
Address of a pointer that will be initialized with the address of a DirectDraw surface representing the current background-depth buffer.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetSceneBackgroundDepth

IDirect3DRMFrame::GetSceneFogColor

Retrieves the fog color of a scene.

D3DCOLOR GetSceneFogColor( );

Return Values

Returns the fog color.

IDirect3DRMFrame::GetSceneFogEnable

Returns whether fog is currently enabled for this scene.

BOOL GetSceneFogEnable( );

Return Values

Returns TRUE if fog is enabled, and FALSE otherwise.

IDirect3DRMFrame::GetSceneFogMode

Returns the current fog mode for this scene.

D3DRMFOGMODE GetSceneFogMode( );

Return Values

Returns a member of the D3DRMFOGMODE enumerated type that specifies the current fog mode.

IDirect3DRMFrame::GetSceneFogParams

Retrieves the current fog parameters for this scene.

HRESULT GetSceneFogParams(
D3DVALUE * *lprvStart*,
D3DVALUE * *lprvEnd*,
D3DVALUE * *lprvDensity*
);

Parameters

lprvStart, lprvEnd, and *lprvDensity*
Addresses of variables that will be the fog start, end, and density values.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::GetSortMode

Retrieves the sorting mode used to process child frames.

D3DRMSORTMODE GetSortMode( );

Return Values

Returns the member of the D3DRMSORTMODE enumerated type that specifies the sorting mode.

See Also

IDirect3DRMFrame::SetSortMode

IDirect3DRMFrame::GetTexture

Retrieves the texture of the given frame.

HRESULT GetTexture(
    LPDIRECT3DRMTEXTURE* *lplpTexture*
    );

Parameters

lplpTexture
Address of the pointer that will be filled with the address of the Direct3DRMTexture object representing the frame's texture.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetTexture

IDirect3DRMFrame::GetTextureTopology

Retrieves the topological properties of a texture when mapped onto objects in the given frame.

HRESULT GetTextureTopology(
BOOL * *lpbWrap_u***,**
BOOL * *lpbWrap_v*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetTextureTopology

IDirect3DRMFrame::GetTransform

Retrieves the local transformation of the frame as a 4×4 affine matrix.

HRESULT GetTransform(
    D3DRMMATRIX4D *rmMatrix*
    );

Parameters

rmMatrix

A D3DRMMATRIX4D array that will be filled with the frame's transformation. Because this is an array, this value is actually an address.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

3D Transformations

IDirect3DRMFrame::GetVelocity

Retrieves the velocity of the frame relative to the given reference frame.

HRESULT GetVelocity(
LPDIRECT3DRMFRAME *lpRef*,
LPD3DVECTOR *lprvVel*,
BOOL *fRotVel*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetVelocity

IDirect3DRMFrame::GetVisuals

Retrieves a list of visuals in the frame.

HRESULT GetVisuals(
    LPDIRECT3DRMVISUALARRAY* *lplpVisuals*
    );

Parameters

lplpVisuals
Address of a pointer to be initialized with a valid Direct3DRMVisualArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::GetZbufferMode

Retrieves the z-buffer mode; that is, whether z-buffering is enabled or disabled.

D3DRMZBUFFERMODE GetZbufferMode( );

Return Values

Returns one of the members of the D3DRMZBUFFERMODE enumerated type.

See Also

IDirect3DRMFrame::SetZbufferMode

IDirect3DRMFrame::InverseTransform

Transforms the vector in the *lprvSrc* parameter in world coordinates to model coordinates, and returns the result in the *lprvDst* parameter.

HRESULT InverseTransform(
    D3DVECTOR **lprvDst*,
    D3DVECTOR **lprvSrc*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::Transform, 3D Transformations

IDirect3DRMFrame::Load

Loads a Direct3DRMFrame object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpArgLTP*
);

Parameters

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 that require special formatting. This parameter can be NULL.

lpArgLTP
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Faces the frame toward the target frame, relative to the given reference frame, locking the rotation by the given constraints.

HRESULT LookAt(
LPDIRECT3DRMFRAME *lpTarget*,
LPDIRECT3DRMFRAME *lpRef*,
D3DRMFRAMECONSTRAINT *rfcConstraint*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::Move

Applies the rotations and velocities for all frames in the given hierarchy.

HRESULT Move(
    D3DVALUE *delta*
    );

Parameters

delta
Amount to change the velocity and rotation.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::SetColor

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

HRESULT SetColor(
    D3DCOLOR *rcColor*
    );

Parameters

rcColor
New color for the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a color key to a Direct3DRMFrameInterpolator object.

See Also

IDirect3DRMFrame::GetColor, IDirect3DRMFrame::SetMaterialMode

IDirect3DRMFrame::SetColorRGB

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

HRESULT SetColorRGB(
D3DVALUE *rvRed*,
D3DVALUE *rvGreen*,
D3DVALUE *rvBlue*
);

Parameters

rvRed, rvGreen, and *rvBlue*
New color for the frame. Each component of the color should be in the range 0 to 1.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add an RGB color key to a Direct3DRMFrameInterpolator object.

See Also

IDirect3DRMFrame::SetMaterialMode

IDirect3DRMFrame::SetMaterialMode

Sets the material mode for a frame. The material mode determines the source of material information for visuals rendered with the frame.

HRESULT SetMaterialMode(
    D3DRMMATERIALMODE *rmmMode*
    );

Parameters

rmmMode
One of the members of the D3DRMMATERIALMODE enumerated type.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::GetMaterialMode

IDirect3DRMFrame::SetOrientation

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*].

HRESULT SetOrientation(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvDx*,
D3DVALUE *rvDy*,
D3DVALUE *rvDz*,
D3DVALUE *rvUx*,
D3DVALUE *rvUy*,
D3DVALUE *rvUz*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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*].

This method is also used to add an orientation key to a Direct3DRMRFrameInterpolator object.

See Also

IDirect3DRMFrame::GetOrientation

IDirect3DRMFrame::SetPosition

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.

HRESULT SetPosition(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a position key to a Direct3DRMFrameInterpolator object.

See Also

IDirect3DRMFrame::GetPosition

IDirect3DRMFrame::SetRotation

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.

HRESULT SetRotation(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*,
D3DVALUE *rvTheta*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified rotation changes the matrix with every render tick, unlike the IDirect3DRMFrame::AddRotation method, which changes the objects in the frame only once.

See Also

IDirect3DRMFrame::AddRotation, IDirect3DRMFrame::GetRotation

IDirect3DRMFrame::SetSceneBackground

Sets the background color of a scene.

HRESULT SetSceneBackground(
    D3DCOLOR *rcColor*
    );

Parameters

rcColor
New color for the background.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a background color key to a Direct3DRMFrameInterpolator object.

IDirect3DRMFrame::SetSceneBackgroundDepth

Specifies a background-depth buffer for a scene.

HRESULT SetSceneBackgroundDepth(
    LPDIRECTDRAWSURFACE *lpImage*
    );

Parameters

lpImage
Address of a DirectDraw surface that will store the new background depth for the scene.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMFrame::GetSceneBackgroundDepth

IDirect3DRMFrame::SetSceneBackgroundImage

Specifies a background image for a scene.

HRESULT SetSceneBackgroundImage(
    LPDIRECT3DRMTEXTURE *lpTexture*
    );

Parameters

lpTexture
Address of a Direct3DRMTexture object that will contain the new background scene.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

Sets the background color of a scene.

HRESULT SetSceneBackgroundRGB(
D3DVALUE *rvRed*,
D3DVALUE *rvGreen*,
D3DVALUE *rvBlue*
);

Parameters

rvRed, rvGreen, and *rvBlue*
New color for the background.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a background RGB color key to a Direct3DRMFrameInterpolator object.

IDirect3DRMFrame::SetSceneFogColor

Sets the fog color of a scene.

HRESULT SetSceneFogColor(
D3DCOLOR *rcColor*
);

Parameters

rcColor
New color for the fog.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a fog color key to a Direct3DRMRFrameInterpolator object.

IDirect3DRMFrame::SetSceneFogEnable

Sets the fog enable state.

HRESULT SetSceneFogEnable(
BOOL *bEnable*
);

Parameters

bEnable
New fog enable state.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame::SetSceneFogMode

Sets the fog mode.

HRESULT SetSceneFogMode(
D3DRMFOGMODE *rfMode*
);

Parameters

rfMode
One of the members of the D3DRMFOGMODE enumerated type, specifying the new fog mode.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::SetSceneFogParams

IDirect3DRMFrame::SetSceneFogParams

Sets the current fog parameters for this scene.

HRESULT SetSceneFogParams(
D3DVALUE *rvStart*,
D3DVALUE *rvEnd*,
D3DVALUE *rvDensity*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a fog parameters key to a Direct3DRMRFrameInterpolator object.

See Also

D3DRMFOGMODE, IDirect3DRMFrame::SetSceneFogMode

IDirect3DRMFrame::SetSortMode

Sets the sorting mode used to process child frames. You can use this method to change the properties of hidden-surface-removal algorithms.

HRESULT SetSortMode(
D3DRMSORTMODE *d3drmSM*
);

Parameters

d3drmSM

One of the members of the D3DRMSORTMODE enumerated type, specifying the sorting mode. The default value is D3DRMSORT_FROMPARENT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::GetSortMode

IDirect3DRMFrame::SetTexture

Sets the texture of the frame.

HRESULT SetTexture(
    LPDIRECT3DRMTEXTURE *lpD3DRMTexture*
    );

Parameters

lpD3DRMTexture
Address of a variable that represents the Direct3DRMTexture object to be used.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMFrame::GetTexture, IDirect3DRMFrame::SetMaterialMode

IDirect3DRMFrame::SetTextureTopology

Defines the topological properties of the texture coordinates across objects in the frame.

HRESULT SetTextureTopology(
BOOL *bWrap_u*,
BOOL *bWrap_v*
);

Parameters

bWrap_u and *bWrap_v*
Variables that are set to TRUE to map the texture in the u- and v-directions, respectively.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::GetTextureTopology

IDirect3DRMFrame::SetVelocity

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.

HRESULT SetVelocity(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*,
BOOL *fRotVel*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::GetVelocity

IDirect3DRMFrame::SetZbufferMode

Sets the z-buffer mode; that is, whether z-buffering is enabled or disabled.

HRESULT SetZbufferMode(
D3DRMZBUFFERMODE *d3drmZBM*
);

Parameters

d3drmZBM

One of the members of the D3DRMZBUFFERMODE enumerated type, specifying the z-buffer mode. The default value is D3DRMZBUFFER_FROMPARENT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::GetZbufferMode

IDirect3DRMFrame::Transform

Transforms the vector in the *lpd3dVSrc* parameter in model coordinates to world coordinates, returning the result in the *lpd3dVDst* parameter.

HRESULT Transform(
D3DVECTOR ***lpd3dVDst*,**
D3DVECTOR ***lpd3dVSrc***
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame::InverseTransform, 3D Transformations

IDirect3DRMFrame2

The **IDirect3DRMFrame2** interface is an extension of the IDirect3DRMFrame interface. **IDirect3DRMFrame2** has new methods that enable using materials, bounding boxes, and axes with frames. **IDirect3DRMFrame2** also has a IDirect3DRMFrame2::RayPick method to calculate the intersections of visuals in the frame with a ray of specified position and direction. In addition, **IDirect3DRMFrame2** has one changed method IDirect3DRMFrame2::AddMoveCallback2.

For a conceptual overview, see IDirect3DRMFrame, IDirect3DRMFrame2, and IDirect3DRMFrameArray Interfaces.

The methods of the **IDirect3DRMFrame2** interface can be organized into the following groups:

Axes	GetAxes GetInheritAxes SetAxes SetInheritAxes
Background	GetSceneBackground GetSceneBackgroundDepth SetSceneBackground SetSceneBackgroundDepth SetSceneBackgroundImage SetSceneBackgroundRGB
Bounding Box	GetBox GetBoxEnable GetHierarchyBox SetBox SetBoxEnable
Color	GetColor SetColor SetColorRGB
Fog	GetSceneFogColor GetSceneFogEnable GetSceneFogMode GetSceneFogParams SetSceneFogColor SetSceneFogEnable SetSceneFogMode SetSceneFogParams
Hierarchies	AddChild DeleteChild GetChildren GetParent GetScene
Lighting	AddLight DeleteLight

	GetLights
Loading	Load
Material	GetMaterial SetMaterial
Material modes	GetMaterialMode SetMaterialMode
Positioning and movement	AddMoveCallback2 AddRotation AddScale AddTranslation DeleteMoveCallback GetOrientation GetPosition GetRotation GetVelocity LookAt Move SetOrientation SetPosition SetQuaternion SetRotation SetVelocity
Ray Picking	RayPick
Sorting	GetSortMode GetZbufferMode SetSortMode SetZbufferMode
Textures	GetTexture GetTextureTopology SetTexture SetTextureTopology
Transformations	AddTransform GetTransform InverseTransform Transform
Visual objects	AddVisual DeleteVisual GetVisuals

The **IDirect3DRMFrame2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef

QueryInterface

Release

In addition, the **IDirect3DRMFrame2** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback

Clone

DeleteDestroyCallback

GetAppData

GetClassName

GetName

SetAppData

SetName

The Direct3DRMFrame2 object is obtained by calling the IDirect3DRM2::CreateFrame method.

IDirect3DRMFrame2::AddChild

Adds a child frame to a frame hierarchy.

HRESULT AddChild(
    LPDIRECT3DRMFRAME *lpD3DRMFrameChild*
    );

Parameters

lpD3DRMFrameChild
Address of the Direct3DRMFrame object that will be added as a child.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

To preserve an object's transformation, use the IDirect3DRMFrame2::GetTransform method to retrieve the object's transformation before using the **AddChild** method. Then reapply the transformation after the frame is added.

See Also

Hierarchies

IDirect3DRMFrame2::AddLight

Adds a light to a frame.

HRESULT AddLight(
    LPDIRECT3DRMLIGHT *lpD3DRMLight*
    );

Parameters

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be added to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::AddMoveCallback2

Adds a callback function for special movement processing.

HRESULT AddMoveCallback2(
D3DRMFRAME2MOVECALLBACK *d3drmFMC*,
VOID * *lpArg*,
DWORD *dwFlags*
);

Parameters

d3drmFMC
Application-defined **D3DRMFRAME2MOVECALLBACK** callback function.

lpArg
Application-defined data to be passed to the callback function.

dwFlags
One of the following values:

D3DRMCALLBACK_PREORDER - the default. When IDirect3DRMFrame2::Move traverses the hierarchy, callbacks for a frame are called before any child frames are traversed.

D3DRMCALLBACK_POSTORDER - When IDirect3DRMFrame2::Move traverses the hierarchy, callbacks for a frame are called after the child frames are traversed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Multiple callbacks on an individual frame are called in the order that the callbacks were created.

See Also

IDirect3DRMFrame2::Move, IDirect3DRMFrame2::DeleteMoveCallback

IDirect3DRMFrame2::AddRotation

Adds a rotation about (*rvX*, *rvY*, *rvZ*) by the number of radians specified in *rvTheta*.

HRESULT AddRotation(
D3DRMCOMBINETYPE *rctCombine***,**
D3DVALUE *rvX***,**
D3DVALUE *rvY***,**
D3DVALUE *rvZ***,**
D3DVALUE *rvTheta*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified rotation changes the matrix only for the frame identified by this IDirect3DRMFrame2 interface. This method changes the objects in the frame only once, unlike IDirect3DRMFrame2::SetRotation, which changes the matrix with every render tick.

See Also

3D Transformations, IDirect3DRMFrame2::SetRotation

IDirect3DRMFrame2::AddScale

Scales a frame's local transformation by (*rvX*, *rvY*, *rvZ*).

HRESULT AddScale(
D3DRMCOMBINETYPE *rctCombine*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified transformation changes the matrix only for the frame identified by this IDirect3DRMFrame2 interface.

See Also

3D Transformations

IDirect3DRMFrame2::AddTransform

Transforms the local coordinates of the frame by the given affine transformation according to the value of the *rctCombine* parameter.

HRESULT AddTransform(
D3DRMCOMBINETYPE *rctCombine*,
D3DRMMATRIX4D *rmMatrix*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Although a 4×4 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 IDirect3DRMFrame2 interface.

See Also

3D Transformations

IDirect3DRMFrame2::AddTranslation

Adds a translation by (*rvX*, *rvY*, *rvZ*) to a frame's local coordinate system.

HRESULT AddTranslation(
D3DRMCOMBINETYPE *rctCombine*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified translation changes the matrix only for the frame identified by this IDirect3DRMFrame2 interface.

See Also

3D Transformations

IDirect3DRMFrame2::AddVisual

Adds a visual object to a frame.

HRESULT AddVisual(
    LPDIRECT3DRMVISUAL *lpD3DRMVisual*
    );

Parameters

lpD3DRMVisual
Address of a variable that represents the Direct3DRMVisual object to be added to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

IDirect3DRMFrame2::DeleteChild

Removes a frame from the hierarchy. If the frame is not referenced, it is destroyed along with any child frames, lights, and meshes.

HRESULT DeleteChild(
    LPDIRECT3DRMFRAME *lpChild*
    );

Parameters

lpChild
Address of a variable that represents the Direct3DRMFrame object to be used as the child.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

Hierarchies

IDirect3DRMFrame2::DeleteLight

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.

HRESULT DeleteLight(
    LPDIRECT3DRMLIGHT *lpD3DRMLight*
    );

Parameters

lpD3DRMLight
Address of a variable that represents the Direct3DRMLight object to be removed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::DeleteMoveCallback

Removes a callback function that performed special movement processing.

HRESULT DeleteMoveCallback(
D3DRMFRAMEMOVECALLBACK *d3drmFMC*,
VOID * *lpArg*
);

Parameters

d3drmFMC
Application-defined D3DRMFRAMEMOVECALLBACK callback function.

lpArg
Application-defined data that was passed to the callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::AddMoveCallback2, IDirect3DRMFrame2::Move

IDirect3DRMFrame2::DeleteVisual

Removes a visual object from a frame, destroying it if it is no longer referenced.

HRESULT DeleteVisual(
    LPDIRECT3DRMVISUAL *lpD3DRMVisual*
    );

Parameters

lpD3DRMVisual
Address of a variable that represents the Direct3DRMVisual object to be removed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::GetAxes

Retrieves the vectors that are aligned with the direction (*rvDx*, *rvDy*, *rvDz*) and up (*rvUx*, *rvUy*, *rvUz*) vectors supplied to the IDirect3DRMFrame2::SetOrientation method.

HRESULT GetAxes(
LPD3DVECTOR *dir*,
LPD3DVECTOR *up*
);

Parameters

dir
The z-axis for the frame. Default is (0,0,1).

up
The y-axis for the frame. Default is (0,1,0).

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method along with IDirect3DRMFrame2::SetAxes helps support both left-handed and right-handed coordinate systems. **SetAxes** allows you to specify that the negative z-axis represents the front of the object.

See Also

IDirect3DRMFrame2::SetAxes, IDirect3DRMFrame2::GetInheritAxes, IDirect3DRMFrame2::SetInheritAxes

IDirect3DRMFrame2::GetBox

Retrieves the bounding box containing a DIRECT3DRMFRAME2 object. The bounding box gives the minimum and maximum model coordinates in each dimension.

HRESULT GetBox(
D3DRMBOX * *lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. Returns D3DRMERR_BOXNOTSET unless a valid bounding box has already been set on the frame. For a list of other possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method supports a bounding box on a frame for hierarchical culling. A valid bounding box must be set on the frame with IDirect3DRMFrame2::SetBox. For a bounding box to be enabled, the IDirect3DRMFrame2::SetBoxEnable method must be called to set the enable flag to TRUE. By default, the box enable flag is FALSE. There is no default bounding box.

See Also

IDirect3DRMFrame2::SetBox, IDirect3DRMFrame2::SetBoxEnable, IDirect3DRMFrame2::GetBoxEnable

IDirect3DRMFrame2::GetBoxEnable

Retrieves the flag that determines whether a bounding box is enabled for this Direct3DRMFrame2 object.

BOOL GetBoxEnable( );

Return Values

Returns TRUE if a bounding box is enabled, or FALSE if it is not enabled.

Remarks

For a bounding box to be enabled, the IDirect3DRMFrame2::SetBoxEnable flag must be called to set the enable flag to TRUE. By default, the box enable flag is FALSE.

See Also

IDirect3DRMFrame2::SetBoxEnable IDirect3DRMFrame2::GetBox, IDirect3DRMFrame2::SetBox

IDirect3DRMFrame2::GetChildren

Retrieves a list of child frames in the form of a Direct3DRMFrameArray object.

HRESULT GetChildren(
    LPDIRECT3DRMFRAMEARRAY* *lplpChildren*
    );

Parameters

lplpChildren
Address of a pointer to be initialized with a valid Direct3DRMFrameArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

DIRECT3DRMFRAMEARRAY, Hierarchies

IDirect3DRMFrame2::GetColor

Retrieves the color of the frame.

D3DCOLOR GetColor( );

Return Values

Returns the color of the Direct3DRMFrame2 object.

See Also

IDirect3DRMFrame2::SetColor

IDirect3DRMFrame2::GetHierarchyBox

Calculates a bounding box to contain all the geometry in the hierarchy rooted in this Direct3DRMFrame2 object.

HRESULT GetHierarchyBox(
D3DRMBOX * *lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetBox, IDirect3DRMFrame2::SetBox, IDirect3DRMFrame2::SetBoxEnable, IDirect3DRMFrame2::GetBoxEnable

IDirect3DRMFrame2::GetInheritAxes

Retrieves the flag that indicates whether the axes for the frame are inherited from the parent frame.

BOOL GetInheritAxes( );

Return Values

Returns TRUE if the frame inherits axes (the default) and FALSE if the frame does not inherit axes.

Remarks

By default, the axes are inherited from the parent. If a frame is set to inherit from the parent and there is no parent, the frame acts as if it inherits from a parent with the default axes (direction=(0,0,1) and up=(0,1,0)).

This method and IDirect3DRMFrame2::SetInheritAxes method allow a single policy for axes to be set at the root of the hierarchy.

See Also

IDirect3DRMFrame2::SetInheritAxes, IDirect3DRMFrame2::GetAxes, IDirect3DRMFrame2::SetAxes

IDirect3DRMFrame2::GetLights

Retrieves a list of lights in the frame in the form of a Direct3DRMLightArray object.

HRESULT GetLights(
    LPDIRECT3DRMLIGHTARRAY* *lplpLights*
    );

Parameters

lplpLights
Address of a pointer to be initialized with a valid Direct3DRMLightArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMLightArray

IDirect3DRMFrame2::GetMaterial

Retrieves the material of the Direct3DRMFrame2 object.

HRESULT GetMaterial(
    LPDIRECT3DRMMATERIAL **lplpMaterial*
    );

Parameters

lplpMaterial

Address of a variable that will be filled with a pointer to the Direct3DRMMaterial object applied to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetMaterial

IDirect3DRMFrame2::GetMaterialMode

Retrieves the material mode of the frame.

D3DRMMATERIALMODE GetMaterialMode( );

Return Values

Returns a member of the D3DRMMATERIALMODE enumerated type that specifies the current material mode.

See Also

IDirect3DRMFrame2::SetMaterialMode

IDirect3DRMFrame2::GetOrientation

Retrieves the orientation of a frame relative to the given reference frame.

HRESULT GetOrientation(
LPDIRECT3DRMFRAME *lpRef*,
LPD3DVECTOR *lprvDir*,
LPD3DVECTOR *lprvUp*
);

Parameters

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 normalized directions of the frame's z-axis and y-axis, respectively.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetOrientation

IDirect3DRMFrame2::GetParent

Retrieves the parent frame of the current frame.

HRESULT GetParent(
    LPDIRECT3DRMRAME * *lplpParent*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::GetPosition

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.

HRESULT GetPosition(
LPDIRECT3DRMFRAME *lpRef*,
LPD3DVECTOR *lprvPos*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetPosition

IDirect3DRMFrame2::GetRotation

Retrieves the rotation of the frame relative to the given reference frame.

HRESULT GetRotation(
LPDIRECT3DRMFRAME *lpRef*,
LPD3DVECTOR *lprvAxis*,
LPD3DVALUE *lprvTheta*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetRotation, Transformations

IDirect3DRMFrame2::GetScene

Retrieves the root frame of the hierarchy containing the given frame.

HRESULT GetScene(
    LPDIRECT3DRMFRAME *lplpRoot*
    );

Parameters

lplpRoot

Address of the pointer that will be filled with the pointer to the Direct3DRMFrame object representing the scene's root frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::GetSceneBackground

Retrieves the background color of a scene.

D3DCOLOR GetSceneBackground( );

Return Values

Returns the color.

IDirect3DRMFrame2::GetSceneBackgroundDepth

Retrieves the current background-depth buffer for the scene.

HRESULT GetSceneBackgroundDepth(
    LPDIRECTDRAWSURFACE * *lplpDDSurface*
    );

Parameters

lplpDDSurface

Address of a pointer that will be initialized with the address of a DirectDraw surface representing the current background-depth buffer.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetSceneBackgroundDepth

IDirect3DRMFrame2::GetSceneFogColor

Retrieves the fog color of a scene.

D3DCOLOR GetSceneFogColor( );

Return Values

Returns the fog color.

IDirect3DRMFrame2::GetSceneFogEnable

Returns whether fog is currently enabled for this scene.

BOOL GetSceneFogEnable( );

Return Values

Returns TRUE if fog is enabled, and FALSE otherwise.

IDirect3DRMFrame2::GetSceneFogMode

Returns the current fog mode for this scene.

D3DRMFOGMODE GetSceneFogMode( );

Return Values

Returns a member of the D3DRMFOGMODE enumerated type that specifies the current fog mode.

IDirect3DRMFrame2::GetSceneFogParams

Retrieves the current fog parameters for this scene.

HRESULT GetSceneFogParams(
D3DVALUE * *lprvStart*,
D3DVALUE * *lprvEnd*,
D3DVALUE * *lprvDensity*
);

Parameters

lprvStart, lprvEnd, and *lprvDensity*
Addresses of variables that will be the fog start, end, and density values.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::GetSortMode

Retrieves the sorting mode used to process child frames.

D3DRMSORTMODE GetSortMode( );

Return Values

Returns the member of the D3DRMSORTMODE enumerated type that specifies the sorting mode.

See Also

IDirect3DRMFrame2::SetSortMode

IDirect3DRMFrame2::GetTexture

Retrieves the texture of the given frame.

HRESULT GetTexture(
    LPDIRECT3DRMTEXTURE* *lplpTexture*
    );

Parameters

lplpTexture
Address of the pointer that will be filled with the address of the Direct3DRMTexture object representing the frame's texture.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetTexture

IDirect3DRMFrame2::GetTextureTopology

Retrieves the topological properties of a texture when mapped onto objects in the given frame.

HRESULT GetTextureTopology(
BOOL * *lpbWrap_u*,
BOOL * *lpbWrap_v*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetTextureTopology

IDirect3DRMFrame2::GetTransform

Retrieves the local transformation of the frame as a 4×4 affine matrix.

HRESULT GetTransform(
    D3DRMMATRIX4D *rmMatrix*
    );

Parameters

rmMatrix

A D3DRMMATRIX4D array that will be filled with the frame's transformation. Because this is an array, this value is actually an address.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

3D Transformations

IDirect3DRMFrame2::GetVelocity

Retrieves the velocity of the frame relative to the given reference frame.

HRESULT GetVelocity(
    LPDIRECT3DRMFRAME *lpRef*,
    LPD3DVECTOR *lprvVel*,
    BOOL *fRotVel*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetVelocity

IDirect3DRMFrame2::GetVisuals

Retrieves a list of visuals in the frame.

HRESULT GetVisuals(
    LPDIRECT3DRMVISUALARRAY* *lplpVisuals*
    );

Parameters

lplpVisuals
Address of a pointer to be initialized with a valid Direct3DRMVisualArray pointer if the call succeeds.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::GetZbufferMode

Retrieves the z-buffer mode; that is, whether z-buffering is enabled or disabled.

D3DRMZBUFFERMODE GetZbufferMode( );

Return Values

Returns one of the members of the D3DRMZBUFFERMODE enumerated type.

See Also

IDirect3DRMFrame2::SetZbufferMode

IDirect3DRMFrame2::InverseTransform

Transforms the vector in the *lprvSrc* parameter in world coordinates to model coordinates, and returns the result in the *lprvDst* parameter.

HRESULT InverseTransform(
    D3DVECTOR ***lprvDst*,**
    D3DVECTOR ***lprvSrc***
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::Transform, 3D Transformations

IDirect3DRMFrame2::Load

Loads a Direct3DRMFrame2 object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpArgLTP*
);

Parameters

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 that require special formatting. This parameter can be NULL.

lpArgLTP
Address of application-defined data passed to the D3DRMLOADTEXTURECALLBACK callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

IDirect3DRMFrame2::LookAt

Faces the frame toward the target frame, relative to the given reference frame, locking the rotation by the given constraints.

HRESULT LookAt(
LPDIRECT3DRMFRAME *lpTarget*,
LPDIRECT3DRMFRAME *lpRef*,
D3DRMFRAMECONSTRAINT *rfcConstraint*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::Move

Applies the rotations and velocities for all frames in the given hierarchy.

HRESULT Move(
    D3DVALUE *delta*
    );

Parameters

delta
Amount to change the velocity and rotation.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::RayPick

Searches the hierarchy starting at this Direct3DRMFrame2 object and calculates the intersections between any visuals and the ray specified by the *dvPosition* and *dvDirection* parameters in the coordinate space specified by the *lpRefFrame* parameter.

HRESULT RayPick(
LPDIRECT3DRMFRAME *lpRefFrame*,
LPD3DRMRAY *ray*,
DWORD *dwFlags*,
LPD3DRMPICKED2ARRAY* *lplpPicked2Array*
);

Parameters

lpRefFrame

Address of the Direct3DRMFrame object that contains the ray.

ray

A pointer to a D3DRMRAY structure that contains two D3DVECTOR structures. The first D3DVECTOR structure is the vector direction of the ray. The second D3DVECTOR structure is the position of the origin of the ray.

dwFlags

Can be one of the following values:

D3DRMRAYPICK_ONLYBOUNDINGBOXES – Only intersections with bounding boxes of the visuals in the hierarchy are returned. Does not check for exact face intersections.

D3DRMRAYPICK_IGNOREFURTHERPRIMITIVES – Only the closest visual that intersects the ray is returned. Ignores visuals further away than the closest one found so far in a search.

D3DRMRAYPICK_INTERPOLATEUV – Interpolate texture coordinates.

D3DRMRAYPICK_INTERPOLATECOLOR – Interpolate color.

D3DRMRAYPICK_INTERPOLATENORMAL – Interpolate normal.

lplpPicked2Array

The address of a pointer to be initialized with a valid pointer to the IDirect3DRMPicked2Array interface. You then call the IDirect3DRMPicked2Array::GetPick method to retrieve a visual object, an IDirect3DRMFrameArray interface, and a D3DRMPICKDESC2 structure. The array of frames is the path through the hierarchy leading to the visual object that intersected the ray. The D3DRMPICKDESC2 structure contains the face and group identifiers, pick position, horizontal and vertical texture coordinates for the vertex, vertex normal, and color of the intersected objects.

If you specify D3DRMRAYPICK_ONLYBOUNDINGBOXES, the texture, normal and color data in the D3DRMPICKDESC2 structure will be invalid.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

There are two kinds of flags: optimization flags and interpolation flags. Optimization flags allow you to limit the search and therefore make it faster. Interpolation flags specify what to interpolate if a primitive is hit. The three interpolation choices are color, normal, and texture coordinates.

The ray is specified in the reference frame coordinate space (pointed to by *lpRefFrame*). If the reference frame is NULL, the ray is specified in world coordinates.

IDirect3DRMFrame2::Save

Saves a Direct3DRMFrame2 object to the specified file.

HRESULT Save(
LPCSTR *lpFilename*,
D3DRMXOFFORMAT *d3dFormat*,
D3DRMSAVEOPTIONS *d3dSaveFlags*
);

Parameters

lpFilename
Address specifying the name of the created file. This file must have a .X file name extension.

d3dFormat
The D3DRMXOF_TEXT value from the D3DRMXOFFORMAT enumerated type.

d3dSaveFlags
Value of the D3DRMSAVEOPTIONS type describing the save options.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::SetAxes

Sets the vectors that define a coordinate space by which the IDirect3DRMFrame2::SetOrientation vectors are transformed.

HRESULT SetAxes(
D3DVALUE *dx*,
D3DVALUE *dy*,
D3DVALUE *dz*,
D3DVALUE *ux*,
D3DVALUE *uy*,
D3DVALUE *uz*
);

Parameters

dx, dy, dz
The z-axis for the frame. Default is (0,0,1).

ux, uy, uz
The y-axis for the frame. Default is (0,1,0).

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method helps support both left-handed and right-handed coordinate systems. This method also allows you to specify that the negative z-axis represents the front of the object.

The IDirect3DRMFrame2::SetOrientation direction (*rvDx*, *rvDy*, *rvDz*) and up (*rvUx*, *rvUy*, *rvUz*) vectors are transformed according to the value of the **SetAxes** vectors.

The axes are inherited by child frames as specified in IDirect3DRMFrame2::SetInheritAxes.

See Also

IDirect3DRMFrame2::GetAxes, IDirect3DRMFrame2::GetInheritAxes, IDirect3DRMFrame2::SetInheritAxes

IDirect3DRMFrame2::SetBox

Sets the box to be used in bounding box testing. In order for the bounding box to be valid, its minimum x must be less than or equal to its maximum x, minimum y must be less than or equal to its maximum y, and minimum z must be less than or equal to its maximum z.

HRESULT SetBox(
D3DRMBOX * *lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that contains the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method supports a bounding box on a frame for hierarchical culling. For a bounding box to be enabled, the IDirect3DRMFrame2::SetBoxEnable method must be called to set the enable flag to TRUE. By default, the box enable flag is FALSE.

See Also

IDirect3DRMFrame2::GetBox, IDirect3DRMFrame2::SetBoxEnable, IDirect3DRMFrame2::GetBoxEnable

IDirect3DRMFrame2::SetBoxEnable

Enables or disables bounding box testing for this Direct3DRMFrame2 object. Bounding box testing cannot be enabled unless a valid bounding box has already been set on the frame.

HRESULT SetBoxEnable(
BOOL *bEnableFlag*
);

Parameters

bEnableFlag
TRUE to enable a bounding box. FALSE if a bounding box is not enabled. Default is FALSE.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

For a bounding box to be enabled, this method must be called to set the enable flag to TRUE. By default, the box enable flag is FALSE.

Bounding box testing is performed as follows: at render time the bounding box is transformed into model space and checked for intersection with the viewing frustum. If all of the box is outside of the viewing frustum, none of the visuals in the frame or in any child frames are rendered. Otherwise, rendering continues as normal.

Enabling bounding box testing with a box of {0,0,0,0} completely prevents a frame from being rendered.

See Also

IDirect3DRMFrame2::GetBoxEnable, IDirect3DRMFrame2::GetBox, IDirect3DRMFrame2::SetBox

IDirect3DRMFrame2::SetColor

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

HRESULT SetColor(
D3DCOLOR *rcColor*
);

Parameters

rcColor
New color for the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a color key to a DIRECT3DRMFRAMEINTERPOLATOR object.

See Also

IDirect3DRMFrame2::GetColor, IDirect3DRMFrame2::SetMaterialMode

IDirect3DRMFrame2::SetColorRGB

Sets the color of the frame. This color is used for meshes in the frame when the D3DRMMATERIALMODE enumerated type is D3DRMMATERIAL_FROMFRAME.

HRESULT SetColorRGB(
D3DVALUE *rvRed*,
D3DVALUE *rvGreen*,
D3DVALUE *rvBlue*
);

Parameters

rvRed, rvGreen, and *rvBlue*
New color for the frame. Each component of the color should be in the range 0 to 1.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add an RGB color key to a DIRECT3DRMFRAMEINTERPOLATOR object.

See Also

IDirect3DRMFrame2::SetMaterialMode

IDirect3DRMFrame2::SetInheritAxes

Specifies whether the axes for the frame are inherited from the parent frame.

HRESULT SetInheritAxes(
BOOL *inherit_from_parent*
);

Parameters

inherit_from_parent

Flag indicating whether the frame should inherit axes from its parent. If TRUE, the frame inherits axes (the default). If FALSE, the frame does not inherit axes.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

By default, the axes are inherited from the parent. If a frame is set to inherit from the parent and there is no parent, the frame acts as if it inherits from a parent with the default axes (direction=(0,0,1)and up=(0,1,0)). This method allows a single policy for axes to be set at the root of the hierarchy.

See Also

IDirect3DRMFrame2::GetInheritAxes, IDirect3DRMFrame2::GetAxes, IDirect3DRMFrame2::SetAxes

IDirect3DRMFrame2::SetMaterial

Sets the material of the Direct3DRMFrame2 object.

HRESULT SetMaterial(
    LPDIRECT3DRMMATERIAL **lplpMaterial*
    );

Parameters

lplpMaterial
Address of the Direct3DRMMaterial object that will be applied to the frame.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetMaterial

IDirect3DRMFrame2::SetMaterialMode

Sets the material mode for a frame. The material mode determines the source of material information for visuals rendered with the frame.

HRESULT SetMaterialMode(
    D3DRMMATERIALMODE *rmmMode*
    );

Parameters

rmmMode
One of the members of the D3DRMMATERIALMODE enumerated type.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetMaterialMode

IDirect3DRMFrame2::SetOrientation

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*].

HRESULT SetOrientation(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvDx*,
D3DVALUE *rvDy*,
D3DVALUE *rvDz*,
D3DVALUE *rvUx*,
D3DVALUE *rvUy*,
D3DVALUE *rvUz*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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*].

This method is also used to add an orientation key to a Direct3DRMFrameInterpolator object.

See Also

IDirect3DRMFrame2::GetOrientation

IDirect3DRMFrame2::SetPosition

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.

HRESULT SetPosition(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a position key to a Direct3DRMFrameInterpolator object.

See Also

IDirect3DRMFrame2::GetPosition

IDirect3DRMFrame2::SetQuaternion

Sets a frame's orientation relative to a reference frame using a unit quaternion.

HRESULT SetQuaternion(
LPDIRECT3DRMFRAME2 *lpRef*,
D3DRMQUATERNION **quat*
)

Parameters

lpRef
Address of a variable that represents the Direct3DRMFrame2 object to be used as the reference.

quat
A D3DRMQUATERNION structure that holds the unit quaternion.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

A quaternion is a four-valued vector that can be used to represent any rotation, and that has properties that are useful when interpolating between orientations. A quaternion is a unit quaternion if s**2 + x**2 + y**2 + z**2 = 1

The function D3DRMQuaternionFromRotation can be used to generate unit quaternions from arbitrary rotation values.

The **SetQuaternion** method is supported by FrameInterpolators. See IDirect3DRMInterpolator Interface for more information about interpolators.

IDirect3DRMFrame2::SetRotation

Sets a frame rotating by the given angle around the given vector at each call to the IDirect3DRM::Tick or IDirect3DRMFrame2::Move method. The direction vector [*rvX, rvY, rvZ*] is defined in the reference frame.

HRESULT SetRotation(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*,
D3DVALUE *rvTheta*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The specified rotation changes the matrix with every render tick, unlike the IDirect3DRMFrame2::AddRotation method, which changes the objects in the frame only once.

See Also

IDirect3DRMFrame2::AddRotation, IDirect3DRMFrame2::GetRotation

IDirect3DRMFrame2::SetSceneBackground

Sets the background color of a scene.

HRESULT SetSceneBackground(
    D3DCOLOR *rcColor*
    );

Parameters

rcColor
New color for the background.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a background color key to a Direct3DRMFrameInterpolator object.

IDirect3DRMFrame2::SetSceneBackgroundDepth

Specifies a background-depth buffer for a scene.

HRESULT SetSceneBackgroundDepth(
    LPDIRECTDRAWSURFACE *lpImage*
    );

Parameters

lpImage
Address of a DirectDraw surface that will store the new background depth for the scene.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMFrame2::GetSceneBackgroundDepth

IDirect3DRMFrame2::SetSceneBackgroundImage

Specifies a background image for a scene.

HRESULT SetSceneBackgroundImage(
    LPDIRECT3DRMTEXTURE *lpTexture*
    );

Parameters

lpTexture
Address of a Direct3DRMTexture object that will contain the new background scene.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

IDirect3DRMFrame2::SetSceneBackgroundRGB

Sets the background color of a scene.

HRESULT SetSceneBackgroundRGB(
D3DVALUE *rvRed*,
D3DVALUE *rvGreen*,
D3DVALUE *rvBlue*
);

Parameters

rvRed, rvGreen, and *rvBlue*
New color for the background.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a background RGB color key to a Direct3DRMFrameInterpolator object.

IDirect3DRMFrame2::SetSceneFogColor

Sets the fog color of a scene.

HRESULT SetSceneFogColor(
D3DCOLOR *rcColor*
);

Parameters

rcColor
New color for the fog.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a fog color key to a Direct3DRMRFrameInterpolator object.

IDirect3DRMFrame2::SetSceneFogEnable

Sets the fog enable state.

HRESULT SetSceneFogEnable(
BOOL *bEnable*
);

Parameters

bEnable
New fog enable state.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMFrame2::SetSceneFogMode

Sets the fog mode.

HRESULT SetSceneFogMode(
D3DRMFOGMODE *rfMode*
);

Parameters

rfMode
One of the members of the D3DRMFOGMODE enumerated type, specifying the new fog mode.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::SetSceneFogParams

IDirect3DRMFrame2::SetSceneFogParams

Sets the current fog parameters for this scene.

HRESULT SetSceneFogParams(
D3DVALUE *rvStart*,
D3DVALUE *rvEnd*,
D3DVALUE *rvDensity*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a fog parameters key to a Direct3DRMRFrameInterpolator object.

See Also

D3DRMFOGMODE, IDirect3DRMFrame2::SetSceneFogMode

IDirect3DRMFrame2::SetSortMode

Sets the sorting mode used to process child frames. You can use this method to change the properties of hidden-surface-removal algorithms.

HRESULT SetSortMode(
D3DRMSORTMODE *d3drmSM*
);

Parameters

d3drmSM

One of the members of the D3DRMSORTMODE enumerated type, specifying the sorting mode. The default value is D3DRMSORT_FROMPARENT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetSortMode

IDirect3DRMFrame2::SetTexture

Sets the texture of the frame.

HRESULT SetTexture(
    LPDIRECT3DRMTEXTURE *lpD3DRMTexture*
    );

Parameters

lpD3DRMTexture
Address of a variable that represents the Direct3DRMTexture object to be used.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMFrame2::GetTexture, IDirect3DRMFrame2::SetMaterialMode

IDirect3DRMFrame2::SetTextureTopology

Defines the topological properties of the texture coordinates across objects in the frame.

HRESULT SetTextureTopology(
BOOL *bWrap_u*,
BOOL *bWrap_v*
);

Parameters

bWrap_u and *bWrap_v*
Variables that are set to TRUE to map the texture in the u- and v-directions, respectively.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetTextureTopology

IDirect3DRMFrame2::SetVelocity

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 IDirect3DRMFrame2::Move method.

HRESULT SetVelocity(
LPDIRECT3DRMFRAME *lpRef*,
D3DVALUE *rvX*,
D3DVALUE *rvY*,
D3DVALUE *rvZ*,
BOOL *fRotVel*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetVelocity

IDirect3DRMFrame2::SetZbufferMode

Sets the z-buffer mode; that is, whether z-buffering is enabled or disabled.

HRESULT SetZbufferMode(
D3DRMZBUFFERMODE *d3drmZBM*
);

Parameters

d3drmZBM

One of the members of the D3DRMZBUFFERMODE enumerated type, specifying the z-buffer mode. The default value is D3DRMZBUFFER_FROMPARENT.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::GetZbufferMode

IDirect3DRMFrame2::Transform

Transforms the vector in the *lpd3dVSrc* parameter in model coordinates to world coordinates, returning the result in the *lpd3dVDst* parameter.

HRESULT Transform(
D3DVECTOR ***lpd3dVDst*,**
D3DVECTOR ***lpd3dVSrc***
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMFrame2::InverseTransform, 3D Transformations

IDirect3DRMInterpolator

Interpolators provide a way of storing actions and applying them to objects with automatic calculation of in-between values. With an interpolator you can blend colors, move objects smoothly between positions, morph meshes, and perform many other transformations.

The **IDirect3DRMInterpolator** interface is a superset of the IDirect3DRMAnimation interface that increases the kinds of object parameters you can animate. While IDirect3DRMAnimation allows animation of an object's position, size and orientation, **IDirect3DRMInterpolator** further enables animation of color, meshes, texture, and material.

For a conceptual overview, see IDirect3DRMInterpolator Overview.

In addition to the standard *IUnknown* and IDirect3DRMObject methods, **IDirect3DRMInterpolator** contains the following methods:

Attaching Objects	AttachObject
	DetachObject
	GetAttachedObjects
Interpolating	GetIndex
	Interpolate
	SetIndex

The **IDirect3DRMInterpolator** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

In addition, **IDirect3DRMInterpolator** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

IDirect3DRMInterpolator::AttachObject

Connects an object to the interpolator.

HRESULT AttachObject(
LPDIRECT3DRMOBJECT *lpD3DRMObject*
)

Parameters

lpD3DRMObject
Address of the Direct3DRMObject object to be attached to the interpolator.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The attached object can be another interpolator or an object of type *x* where the interpolator is of type *x*Interpolator. For example, a Viewport can be attached to a ViewportInterpolator. The interpolator types are:

- FrameInterpolator
- LightInterpolator
- MaterialInterpolator
- MeshInterpolator
- TextureInterpolator
- ViewportInterpolator

IDirect3DRMInterpolator::DetachObject

Detaches an object from the interpolator.

HRESULT DetachObject(
LPDIRECT3DRMOBJECT *lpD3DRMObject*
)

Parameters

lpD3DRMObject
Address of the Direct3DRMObject object to be detached from the interpolator.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMInterpolator::GetAttachedObjects

Returns an array of objects currently attached to the interpolator.

HRESULT GetAttachedObjects(
LPDIRECT3DRMOBJECTARRAY *lpD3DRMObjectArray*
)

Parameters

lpD3DRMObjectArray
Address of an IDirect3DRMObjectArray object containing the Direct3DRMObject objects currently attached to the interpolator.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMInterpolator::GetIndex

Retrieves the interpolator's current internal index (time).

D3DVALUE GetIndex( )

Return Values

Returns a **D3DVALUE** that contains the interpolator's current internal index.

Remarks

Every key stored in an interpolator has an index value. When a key is recorded (by calling a method), the key is stamped with the current interpolator index value. The key's index value does not change after being stamped.

IDirect3DRMInterpolator::Interpolate

Generates a series of actions by interpolating between keys stored in the interpolator. The actions are then applied to the specified object. If no object is specified, the actions are applied to the currently attached objects.

HRESULT Interpolate(
D3DVALUE *d3dVal*,
LPDIRECT3DRMOBJECT *lpD3DRMObject*,
D3DRMINTERPOLATIONOPTIONS *d3drmInterpFlags*
)

Parameters

d3dVal

A D3DVALUE that contains the interpolator's current internal index.

lpD3DRMObject

Address of the Direct3DRMObject object which will be assigned interpolated values for all properties stored in the interpolator. Can be NULL, in which case the property values of all attached objects will be set to interpolated values.

d3drmInterpFlags

One of more flags that control the kind of interpolation done. Possible values are:

D3DRMINTERPOLATION_CLOSED
D3DRMINTERPOLATION_LINEAR
D3DRMINTERPOLATION_NEAREST
D3DRMINTERPOLATION_OPEN
D3DRMINTERPOLATION_SLERPNORMALS
D3DRMINTERPOLATION_SPLINE
D3DRMINTERPOLATION_VERTEXCOLOR

See Interpolation Options for a description of these options.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMInterpolator::SetIndex

Sets the interpolator's internal index (time) to the specified value. If other interpolators are attached to the interpolator, this method recursively synchronizes their indices to the same value.

HRESULT SetIndex(
D3DVALUE *d3dVal*
)

Parameters

d3dVal
The time to set for the interpolator's internal index.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Every key stored in an interpolator has an index value. When a key is recorded (by calling a method), the key is stamped with the current interpolator index value. The key's index value does not change after being stamped.

IDirect3DRMLight

Applications use the methods of the **IDirect3DRMLight** interface to interact with light objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMLight and IDirect3DRMLightArray Interfaces.

The methods of the **IDirect3DRMLight** interface can be organized into the following groups:

Attenuation	GetConstantAttenuation GetLinearAttenuation GetQuadraticAttenuation SetConstantAttenuation SetLinearAttenuation SetQuadraticAttenuation
Color	GetColor SetColor SetColorRGB
Enable frames	GetEnableFrame SetEnableFrame
Light types	GetType SetType
Range	GetRange SetRange
Spotlight options	GetPenumbra GetUmbra SetPenumbra SetUmbra

The **IDirect3DRMLight** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::GetColor

Retrieves the color of the current Direct3DRMLight object.

D3DCOLOR GetColor( );

Return Values

Returns the color.

See Also

IDirect3DRMLight::SetColor

IDirect3DRMLight::GetConstantAttenuation

Retrieves the constant attenuation factor for the Direct3DRMLight object.

D3DVALUE GetConstantAttenuation( );

Return Values

Returns the constant attenuation value.

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

See Also

IDirect3DRMLight::SetConstantAttenuation

IDirect3DRMLight::GetEnableFrame

Retrieves the enable frame for a light.

HRESULT GetEnableFrame(
    LPDIRECT3DRMFRAME * *lplpEnableFrame*
    );

Parameters

lplpEnableFrame
Address of a pointer that will contain the enable frame for the current Direct3DRMFrame object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMLight::SetEnableFrame

IDirect3DRMLight::GetLinearAttenuation

Retrieves the linear attenuation factor for a light.

D3DVALUE GetLinearAttenuation( );

Return Values

Returns the linear attenuation value.

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

See Also

IDirect3DRMLight::SetLinearAttenuation

IDirect3DRMLight::GetPenumbra

Retrieves the penumbra angle of a spotlight.

D3DVALUE GetPenumbra( );

Return Values

Returns the penumbra value.

See Also

IDirect3DRMLight::SetPenumbra

IDirect3DRMLight::GetQuadraticAttenuation

Retrieves the quadratic attenuation factor for a light.

D3DVALUE GetQuadraticAttenuation( );

Return Values

Returns the quadratic attenuation value.

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

See Also

IDirect3DRMLight::SetQuadraticAttenuation

IDirect3DRMLight::GetRange

Retrieves the range of the current Direct3DRMLight object.

D3DVALUE GetRange( );

Return Values

Returns a value describing the range.

See Also

IDirect3DRMLight::SetRange

IDirect3DRMLight::GetType

Retrieves the type of a given light.

D3DRMLIGHTTYPE GetType( );

Return Values

Returns one of the members of the D3DRMLIGHTTYPE enumerated type.

See Also

IDirect3DRMLight::SetType

IDirect3DRMLight::GetUmbra

Retrieves the umbra angle of the Direct3DRMLight object.

D3DVALUE GetUmbra( );

Return Values

Returns the umbra angle.

See Also

IDirect3DRMLight::SetUmbra

IDirect3DRMLight::SetColor

Sets the color of the given light.

HRESULT SetColor(
    D3DCOLOR *rcColor*
    );

Parameters

rcColor
New color for the light.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a color key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetColor

IDirect3DRMLight::SetColorRGB

Sets the color of the given light.

HRESULT SetColorRGB(
D3DVALUE *rvRed*,
D3DVALUE *rvGreen*,
D3DVALUE *rvBlue*
);

Parameters

rvRed, rvGreen, and *rvBlue*
New color for the light.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add an RGB color key to a Direct3DRMLightInterpolator object.

IDirect3DRMLight::SetConstantAttenuation

Sets the constant attenuation factor for a light.

HRESULT SetConstantAttenuation(
D3DVALUE *rvAtt*
);

Parameters

rvAtt
New constant attenuation factor.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

This method is also used to add a constant attenuation key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetConstantAttenuation

IDirect3DRMLight::SetEnableFrame

Sets the enable frame for a light.

HRESULT SetEnableFrame(
    LPDIRECT3DRMFRAME *lpEnableFrame*
    );

Parameters

lpEnableFrame
Address of the light's enable frame. Child frames of this frame are also enabled for this light source.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMLight::GetEnableFrame

IDirect3DRMLight::SetLinearAttenuation

Sets the linear attenuation factor for a light.

HRESULT SetLinearAttenuation(
D3DVALUE *rvAtt*
);

Parameters

rvAtt
New linear attenuation factor.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

This method is also used to add a linear attenuation key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetLinearAttenuation

IDirect3DRMLight::SetPenumbra

Sets the angle of the penumbra cone.

HRESULT SetPenumbra(
D3DVALUE *rvAngle*
);

Parameters

rvAngle

New penumbra angle. This angle must be greater than or equal to the angle of the umbra. If you 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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a penumbra angle key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetPenumbra

IDirect3DRMLight::SetQuadraticAttenuation

Sets the quadratic attenuation factor for a light.

HRESULT SetQuadraticAttenuation(
D3DVALUE *rvAtt*
);

Parameters

rvAtt
New quadratic attenuation factor.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Remarks

Lights that have a location (are not infinitely far away) can have attenuation factors to calculate the attenuation of the light based on distance from the light.

The formula for the total attenuation factor is:

1 / [*constant_attenuation_factor* + distance * *linear_attenuation_factor* + (distance**2) * *quadratic_attenuation_factor*]

The total attenuation factor cannot be greater than 1. When attenuation factors are not provided, the default values are 1.0 for the constant attenuation factor, 0.0 for the linear attenuation factor, and 0.0 for the quadratic attenuation factor.

This method is also used to add a quadratic attenuation key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetQuadraticAttenuation

IDirect3DRMLight::SetRange

Sets the range of a spot light. The light affects objects that are within the range only.

HRESULT SetRange(
    D3DVALUE *rvRange*
    );

Parameters

rvRange
New range. The default value is 256.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The range property is for spotlights only. This method is also used to add a range key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetRange

IDirect3DRMLight::SetType

Changes the light's type.

HRESULT SetType(
    D3DRMLIGHTTYPE *d3drmtType*
    );

Parameters

d3drmtType
New light type specified by one of the members of the D3DRMLIGHTTYPE enumerated type.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMLight::GetType

IDirect3DRMLight::SetUmbra

Sets the angle of the umbra cone.

HRESULT SetUmbra(
D3DVALUE *rvAngle*
);

Parameters

rvAngle

New umbra angle. This angle must be less than or equal to the angle of the penumbra. If you 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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add an umbra angle key to a Direct3DRMLightInterpolator object.

See Also

IDirect3DRMLight::GetUmbra

IDirect3DRMMaterial

Applications use the methods of the **IDirect3DRMMaterial** interface to interact with material objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMMaterial Interface.

The methods of the **IDirect3DRMMaterial** interface can be organized into the following groups:

Emission	GetEmissive SetEmissive
Power for specular exponent	GetPower SetPower
Specular	GetSpecular SetSpecular

The **IDirect3DRMMaterial** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::GetEmissive

Retrieves the setting for the emissive property of a material. The setting of this property is the color and intensity of the light the object emits.

HRESULT GetEmissive(
D3DVALUE **lpr*,
D3DVALUE **lpg*,
D3DVALUE **lpb*
);

Parameters

lpr, *lpg*, and *lpb*
Addresses that will contain the red, green, and blue components of the emissive color when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMaterial::SetEmissive

IDirect3DRMMaterial::GetPower

Retrieves the power used for the specular exponent in the given material.

D3DVALUE GetPower( );

Return Values

Returns the value specifying the power of the specular exponent.

See Also

IDirect3DRMMaterial::SetPower

IDirect3DRMMaterial::GetSpecular

Retrieves the color of the specular highlights of a material.

HRESULT GetSpecular(
D3DVALUE **lpr*,
D3DVALUE **lpg*,
D3DVALUE **lpb*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMaterial::SetSpecular

IDirect3DRMMaterial::SetEmissive

Sets the emissive property of a material.

HRESULT SetEmissive(
D3DVALUE *r*,
D3DVALUE *g*,
D3DVALUE *b*
);

Parameters

r, *g*, and *b*
Red, green, and blue components of the emissive color.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add an emissive property key to a Direct3DRMMaterialInterpolator object.

See Also

IDirect3DRMMaterial::GetEmissive

IDirect3DRMMaterial::SetPower

Sets the power used for the specular exponent in a material.

HRESULT SetPower(
    D3DVALUE *rvPower*
    );

Parameters

rvPower
New specular exponent.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a spectral power key to a Direct3DRMMaterialInterpolator object.

See Also

IDirect3DRMMaterial::GetPower

IDirect3DRMMaterial::SetSpecular

Sets the color of the specular highlights for a material.

HRESULT SetSpecular(
D3DVALUE *r*,
D3DVALUE *g*,
D3DVALUE *b*
);

Parameters

r, *g*, and *b*
Red, green, and blue components of the color of the specular highlights.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a spectral color key to a Direct3DRMMaterialInterpolator object.

See Also

IDirect3DRMMaterial::GetSpecular

IDirect3DRMMesh

Applications use the methods of the **IDirect3DRMMesh** interface to interact with groups of meshes. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMMesh, IDirect3DRMMeshBuilder, and IDirect3DRMMeshBuilder2 Interfaces.

The methods of the **IDirect3DRMMesh** interface can be organized into the following groups:

Color	GetGroupColor SetGroupColor SetGroupColorRGB
Creation and information	AddGroup GetBox GetGroup GetGroupCount
Materials	GetGroupMaterial SetGroupMaterial
Miscellaneous	Scale Translate
Rendering quality	GetGroupQuality SetGroupQuality
Texture mapping	GetGroupMapping SetGroupMapping
Textures	GetGroupTexture SetGroupTexture
Vertex positions	GetVertices SetVertices

The **IDirect3DRMMesh** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Groups a collection of faces and retrieves an identifier for the group. The first group added to a mesh always has index 0. The index of each successive group increases by 1.

HRESULT AddGroup(
unsigned *vCount*,
unsigned *fCount*,
unsigned *vPerFace*,
unsigned **fData*,
D3DRMGROUPINDEX **returnId*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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::GetBox

Retrieves the bounding box containing a Direct3DRMMesh object. The bounding box gives the minimum and maximum model coordinates in each dimension.

HRESULT GetBox(
    D3DRMBOX * *lpD3DRMBox*
    );

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMesh::GetGroup

Retrieves the data associated with a specified group.

HRESULT GetGroup(
D3DRMGROUPINDEX *id*,
unsigned **vCount*,
unsigned **fCount*,
unsigned **vPerFace*,
DWORD **fDataSize*,
unsigned **fData*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMesh::GetGroupColor

Retrieves the color for a group.

D3DCOLOR GetGroupColor(
D3DRMGROUPINDEX *id*
);

Parameters

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

Return Values

Returns a D3DCOLOR variable specifying the color if successful, or zero otherwise.

See Also

IDirect3DRMMesh::SetGroupColor, IDirect3DRMMesh::SetGroupColorRGB

IDirect3DRMMesh::GetGroupCount

Retrieves the number of groups for a given Direct3DRMMesh object.

unsigned GetGroupCount( );

Return Values

Returns the number of groups if successful, or zero otherwise.

IDirect3DRMMesh::GetGroupMapping

Returns a description of how textures are mapped to a group in a Direct3DRMMesh object.

D3DRMMAPPING GetGroupMapping(
D3DRMGROUPINDEX *id*
);

Parameters

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

Return Values

Returns one of the D3DRMMAPPING values describing how textures are mapped to a group, if successful. Returns zero otherwise.

See Also

IDirect3DRMMesh::SetGroupMapping

IDirect3DRMMesh::GetGroupMaterial

Retrieves a pointer to the material associated with a group in a Direct3DRMMesh object.

HRESULT GetGroupMaterial(
D3DRMGROUPINDEX *id*,
LPDIRECT3DRMMATERIAL **returnPtr*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::SetGroupMaterial

IDirect3DRMMesh::GetGroupQuality

Retrieves the rendering quality for a specified group in a Direct3DRMMesh object.

D3DRMRENDERQUALITY GetGroupQuality(
D3DRMGROUPINDEX *id*
);

Parameters

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

Return Values

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.

See Also

IDirect3DRMMesh::SetGroupQuality

IDirect3DRMMesh::GetGroupTexture

Retrieves an address of the texture associated with a group in a Direct3DRMMesh object.

HRESULT GetGroupTexture(
D3DRMGROUPINDEX *id*,
LPDIRECT3DRMTEXTURE **returnPtr*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::SetGroupTexture

IDirect3DRMMesh::GetVertices

Retrieves vertex information for a specified group in a Direct3DRMMesh object.

HRESULT GetVertices(
D3DRMGROUPINDEX *id*,
DWORD *index*,
DWORD *count*,
D3DRMVERTEX **returnPtr*
);

Parameters

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. To retrieve the number of vertices in a group, call the IDirect3DRMMesh::GetGroup method.

returnPtr

Array of **D3DRMVERTEX** structures that will contain the vertex information (vertex positions, colors, texture coordinates, and so on) when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::GetGroup, IDirect3DRMMesh::SetVertices

IDirect3DRMMesh::Scale

Scales a Direct3DRMMesh object by the given scaling factors, parallel to the x-, y-, and z-axes in model coordinates.

HRESULT Scale(
D3DVALUE *sx*,
D3DVALUE *sy*,
D3DVALUE *sz*
);

Parameters

sx, sy, and *sz*
Scaling factors that are applied along the x-, y-, and z-axes.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMesh::SetGroupColor

Sets the color of a group in a Direct3DRMMesh object.

HRESULT SetGroupColor(
D3DRMGROUPINDEX *id*,
D3DCOLOR *value*
);

Parameters

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

value

Color of the group.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a group color key to a Direct3DRMMeshInterpolator object.

See Also

IDirect3DRMMesh::GetGroupColor, IDirect3DRMMesh::SetGroupColorRGB

IDirect3DRMMesh::SetGroupColorRGB

Sets the color of a group in a Direct3DRMMesh object, using individual RGB values.

HRESULT SetGroupColorRGB(
D3DRMGROUPINDEX *id*,
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a group RGB key to a Direct3DRMMeshInterpolator object.

See Also

IDirect3DRMMesh::GetGroupColor, IDirect3DRMMesh::SetGroupColor

IDirect3DRMMesh::SetGroupMapping

Sets the mapping for a group in a Direct3DRMMesh object. The mapping controls how textures are mapped to a surface.

HRESULT SetGroupMapping(
D3DRMGROUPINDEX *id*,
D3DRMMAPPING *value*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::GetGroupMapping

IDirect3DRMMesh::SetGroupMaterial

Sets the material associated with a group in a Direct3DRMMesh object.

HRESULT SetGroupMaterial(
D3DRMGROUPINDEX *id*,
LPDIRECT3DRMMATERIAL *value*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::GetGroupMaterial

IDirect3DRMMesh::SetGroupQuality

Sets the rendering quality for a specified group in a Direct3DRMMesh object.

HRESULT SetGroupQuality(
D3DRMGROUPINDEX *id*,
D3DRMRENDERQUALITY *value*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::GetGroupQuality

IDirect3DRMMesh::SetGroupTexture

Sets the texture associated with a group in a Direct3DRMMesh object.

HRESULT SetGroupTexture(
D3DRMGROUPINDEX *id*,
LPDIRECT3DRMTEXTURE *value*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMesh::GetGroupTexture

IDirect3DRMMesh::SetVertices

Sets the vertex positions for a specified group in a Direct3DRMMesh object.

HRESULT SetVertices(
D3DRMGROUPINDEX *id*,
unsigned *index*,
unsigned *count*,
D3DRMVERTEX **values*
);

Parameters

id

Identifier of the group. This identifier must have been produced by using the IDirect3DRMMesh::AddGroup method.

index

Index of the first vertex in the mesh group to be modified.

count

Number of consecutive vertices to set. The *values* array must contain at least *count* elements.

values

Array of D3DRMVERTEX structures specifying the vertex positions to be set.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Vertices are local to the group. If an 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.

This method is also used to add a vertex position key to a Direct3DRMMeshInterpolator object.

See Also

IDirect3DRMMesh::GetVertices

IDirect3DRMMesh::Translate

Adds the specified offsets to the vertex positions of a Direct3DRMMesh object.

HRESULT Translate(
D3DVALUE *tx*,
D3DVALUE *ty*,
D3DVALUE *tz*
);

Parameters

tx, ty, and *tz*
Offsets that are added to the x-, y-, and z-coordinates respectively of each vertex position.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder

Applications use the methods of the **IDirect3DRMMeshBuilder** interface to interact with mesh objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMMesh, IDirect3DRMMeshBuilder, and IDirect3DRMMeshBuilder2 Interfaces.

The methods of the **IDirect3DRMMeshBuilder** interface can be organized into the following groups:

Color	GetColorSource SetColor SetColorRGB SetColorSource
Creation and information	GetBox
Faces	AddFace AddFaces CreateFace GetFaceCount GetFaces
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

The **IDirect3DRMMeshBuilder** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Adds a face to a Direct3DRMMeshBuilder object.

HRESULT AddFace(
    LPDIRECT3DRMFACE *lpD3DRMFace*
    );

Parameters

lpD3DRMFace
Address of the face being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Any one face can exist in only one mesh at a time.

IDirect3DRMMeshBuilder::AddFaces

Adds faces to a Direct3DRMMeshBuilder object.

HRESULT AddFaces(
DWORD *dwVertexCount*,
D3DVECTOR * *lpD3DVertices*,
DWORD *normalCount*,
D3DVECTOR **lpNormals*,
DWORD **lpFaceData*,
LPDIRECT3DRMFACEARRAY* *lplpD3DRMFaceArray*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::AddFrame

Adds the contents of a frame to a Direct3DRMMeshBuilder object.

HRESULT AddFrame(
    LPDIRECT3DRMFRAME *lpD3DRMFrame*
    );

Parameters

lpD3DRMFrame
Address of the frame whose contents are being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The source frame is not modified or referenced by this operation.

IDirect3DRMMeshBuilder::AddMesh

Adds a mesh to a Direct3DRMMeshBuilder object.

HRESULT AddMesh(
    LPDIRECT3DRMMESH *lpD3DRMMesh*
    );

Parameters

lpD3DRMMesh
Address of the mesh being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::AddMeshBuilder

Adds the contents of a Direct3DRMMeshBuilder object to another Direct3DRMMeshBuilder object.

HRESULT AddMeshBuilder(
LPDIRECT3DRMMESHBUILDER *lpD3DRMMeshBuild*
);

Parameters

lpD3DRMMeshBuild
Address of the Direct3DRMMeshBuilder object whose contents are being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The source Direct3DRMMeshBuilder object is not modified or referenced by this operation.

IDirect3DRMMeshBuilder::AddNormal

Adds a normal to a Direct3DRMMeshBuilder object.

int AddNormal(
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

x, y, and *z*
The x, y, and z components of the direction of the new normal.

Return Values

Returns the index of the normal.

IDirect3DRMMeshBuilder::AddVertex

Adds a vertex to a Direct3DRMMeshBuilder object.

int AddVertex(
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

x, y, and *z*
The x, y, and z components of the position of the new vertex.

Return Values

Returns the index of the vertex.

IDirect3DRMMeshBuilder::CreateFace

Creates a new face with no vertices and adds it to a Direct3DRMMeshBuilder object.

HRESULT CreateFace(
    LPDIRECT3DRMFACE* *lplpD3DRMFace*
    );

Parameters

lplpD3DRMFace

Address of a pointer to an IDirect3DRMFace interface that will be filled with a pointer to the face that was created.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::CreateMesh

Creates a new mesh from a Direct3DRMMeshBuilder object.

HRESULT CreateMesh(
    LPDIRECT3DRMMESH* *lplpD3DRMMesh*
    );

Parameters

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::GenerateNormals

Processes the Direct3DRMMeshBuilder object and generates vertex normals that are the average of each vertex's adjoining face normals.

HRESULT GenerateNormals( );

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Averaging the normals of back-to-back faces produces a zero normal.

IDirect3DRMMeshBuilder::GetBox

Retrieves the bounding box containing a Direct3DRMMeshBuilder object. The bounding box gives the minimum and maximum model coordinates in each dimension.

HRESULT GetBox(
D3DRMBOX **lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::GetColorSource

Retrieves the color source of a Direct3DRMMeshBuilder object. The color source can be either a face or a vertex.

D3DRMCOLORSOURCE GetColorSource( );

Return Values

Returns a member of the D3DRMCOLORSOURCE enumerated type.

See Also

IDirect3DRMMeshBuilder::SetColorSource

IDirect3DRMMeshBuilder::GetFaceCount

Retrieves the number of faces in a Direct3DRMMeshBuilder object.

int GetFaceCount( );

Return Values

Returns the number of faces.

IDirect3DRMMeshBuilder::GetFaces

Retrieves the faces of a Direct3DRMMeshBuilder object.

HRESULT GetFaces(
    LPDIRECT3DRMFACEARRAY* *lplpD3DRMFaceArray*
    );

Parameters

lplpD3DRMFaceArray
Address of a pointer to an IDirect3DRMFaceArray interface that is filled with an address of the faces.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::GetPerspective

Determines whether perspective correction is on for a Direct3DRMMeshBuilder object.

BOOL GetPerspective( );

Return Values

Returns TRUE if perspective correction is on, or FALSE otherwise.

IDirect3DRMMeshBuilder::GetQuality

Retrieves the rendering quality of a Direct3DRMMeshBuilder object.

D3DRMRENDERQUALITY GetQuality( );

Return Values

Returns a member of the D3DRMRENDERQUALITY enumerated type that specifies the rendering quality of the mesh.

See Also

IDirect3DRMMeshBuilder::SetQuality

IDirect3DRMMeshBuilder::GetTextureCoordinates

Retrieves the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT GetTextureCoordinates(
    DWORD *index*,
    D3DVALUE **lpU*,
    D3DVALUE **lpV*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder::SetTextureCoordinates

IDirect3DRMMeshBuilder::GetVertexColor

Retrieves the color of a specified vertex in a Direct3DRMMeshBuilder object.

D3DCOLOR GetVertexColor(
DWORD *index*
);

Parameters

index
Index of the vertex.

Return Values

Returns the color.

See Also

IDirect3DRMMeshBuilder::SetVertexColor

IDirect3DRMMeshBuilder::GetVertexCount

Retrieves the number of vertices in a Direct3DRMMeshBuilder object.

int GetVertexCount( );

Return Values

Returns the number of vertices.

IDirect3DRMMeshBuilder::GetVertices

Retrieves the vertices, normals, and face data for a Direct3DRMMeshBuilder object.

HRESULT GetVertices(
DWORD **vcount*,
D3DVECTOR **vertices*,
DWORD **ncount*,
D3DVECTOR **normals*,
DWORD **face_data_size*,
DWORD **face_data*
);

Parameters

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. If this parameter is NULL, the method returns the number of vertices in the *vcount* parameter.

ncount
Address of a variable that will contain the number of normals.

normals
Address of an array of **D3DVECTOR** structures that will contain the normals for the Direct3DRMMeshBuilder object. If this parameter is NULL, the method returns the number of normals in the *ncount* parameter.

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 IDirect3DRMMeshBuilder::AddFaces 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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::Load

Loads a Direct3DRMMeshBuilder object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpvArg*
);

Parameters

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 that require special formatting. This parameter can be NULL.

lpvArg
Address of application-defined data passed to the **D3DRMLOADTEXTURECALLBACK** callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

By default, this method loads the first mesh from the source specified in the *lpvObjSource* parameter.

IDirect3DRMMeshBuilder::ReserveSpace

Reserves space within a Direct3DRMMeshBuilder object for the specified number of vertices, normals, and faces. This allows the system to use memory more efficiently.

HRESULT ReserveSpace(
DWORD *vertexCount*,
DWORD *normalCount*,
DWORD *faceCount*
);

Parameters

vertexCount, normalCount, and *faceCount*
Number of vertices, normals, and faces to allocate space for.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::Save

Saves a Direct3DRMMeshBuilder object.

HRESULT Save(
const char * *lpFilename*,
D3DRMXOFFORMAT *d3drmXOFFormat*,
D3DRMSAVEOPTIONS *d3drmSOContents*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::Scale

Scales a Direct3DRMMeshBuilder object by the given scaling factors, parallel to the x-, y-, and z-axes in model coordinates.

HRESULT Scale(
D3DVALUE *sx*,
D3DVALUE *sy*,
D3DVALUE *sz*
);

Parameters

sx, sy, and *sz*
Scaling factors that are applied along the x-, y-, and z-axes.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetColor

Sets all the faces of a Direct3DRMMeshBuilder object to a given color.

HRESULT SetColor(
    D3DCOLOR *color*
    );

Parameters

color
Color of the faces.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetColorRGB

Sets all the faces of a Direct3DRMMeshBuilder object to a given color.

HRESULT SetColorRGB(
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

red, green, and *blue*
Red, green, and blue components of the color.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetColorSource

Sets the color source of a Direct3DRMMeshBuilder object.

HRESULT SetColorSource(
    D3DRMCOLORSOURCE *source*
    );

Parameters

source
Member of the D3DRMCOLORSOURCE enumerated type that specifies the new color source to use.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder::GetColorSource

IDirect3DRMMeshBuilder::SetMaterial

Sets the material of all the faces of a Direct3DRMMeshBuilder object.

HRESULT SetMaterial(
    LPDIRECT3DRMMATERIAL *lpIDirect3DRMmaterial*
    );

Parameters

lpIDirect3DRMmaterial
Address of IDirect3DRMMaterial interface for the Direct3DRMMeshBuilder object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetNormal

Sets the normal vector of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT SetNormal(
DWORD *index*,
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetPerspective

Enables or disables perspective-correct texture-mapping for a Direct3DRMMeshBuilder object.

HRESULT SetPerspective(
BOOL *perspective*
);

Parameters

perspective
Specify TRUE if the mesh should be texture-mapped with perspective correction, or FALSE otherwise.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetQuality

Sets the rendering quality of a Direct3DRMMeshBuilder object.

HRESULT SetQuality(
D3DRMRENDERQUALITY *quality*
);

Parameters

quality

Member of the D3DRMRENDERQUALITY enumerated type that specifies the new rendering quality to use.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

An object's quality has three components: shade mode (flat or Gouraud, Phong is not yet implemented and will default to Gouraud shading), lighting type (on or off), and fill mode (point, wire-frame or solid).

You can set the quality of a device with IDirect3DRMDevice::SetQuality. By default it is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMMeshBuilder object with the **SetQuality** method. By default, a Direct3DRMMeshBuilder object's quality is D3DRMRENDER_GOURAUD (Gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

See Also

IDirect3DRMMeshBuilder::GetQuality

IDirect3DRMMeshBuilder::SetTexture

Sets the texture of all the faces of a Direct3DRMMeshBuilder object.

HRESULT SetTexture(
    LPDIRECT3DRMTEXTURE *lpD3DRMTexture*
    );

Parameters

lpD3DRMTexture
Address of the required Direct3DRMTexture object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetTextureCoordinates

Sets the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT SetTextureCoordinates(
DWORD *index*,
D3DVALUE *u*,
D3DVALUE *v*
);

Parameters

index
Index of the vertex to be set.

u and *v*
Texture coordinates to assign to the specified mesh vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder::GetTextureCoordinates

IDirect3DRMMeshBuilder::SetTextureTopology

Sets the texture topology of a Direct3DRMMeshBuilder object.

HRESULT SetTextureTopology(
BOOL *cylU*,
BOOL *cylV*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetVertex

Sets the position of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT SetVertex(
DWORD *index*,
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::SetVertexColor

Sets the color of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT SetVertexColor(
DWORD *index*,
D3DCOLOR *color*
);

Parameters

index
Index of the vertex to be set.

color
Color to assign to the specified vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder::GetVertexColor

IDirect3DRMMeshBuilder::SetVertexColorRGB

Sets the color of a specified vertex in a Direct3DRMMeshBuilder object.

HRESULT SetVertexColorRGB(
DWORD *index*,
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder::Translate

Adds the specified offsets to the vertex positions of a Direct3DRMMeshBuilder object.

HRESULT Translate(
D3DVALUE *tx*,
D3DVALUE *ty*,
D3DVALUE *tz*
);

Parameters

tx, ty, and *tz*
Offsets that are added to the x-, y-, and z-coordinates respectively of each vertex position.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2

Applications use the methods of the **IDirect3DRMMeshBuilder2** interface to interact with mesh objects. **IDirect3DRMMeshBuilder2** has the same functionality as IDirect3DRMMeshBuilder and in addition has extended the IDirect3DRMMeshBuilder2::GenerateNormals2 method to give greater control over the mesh normals generated. **IDirect3DRMMeshBuilder2** also allows you to retrieve a single mesh face with IDirect3DRMMeshBuilder2::GetFace.

This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMMesh, IDirect3DRMMeshBuilder, and IDirect3DRMMeshBuilder2 Interfaces.

The methods of the **IDirect3DRMMeshBuilder2** interface can be organized into the following groups:

Color	GetColorSource
	SetColor
	SetColorRGB
	SetColorSource
Creation	GetBox
Faces	AddFace
	AddFaces
	CreateFace
	GetFaceCount
	GetFace
	GetFaces
Loading	Load
Meshes	AddMesh
	CreateMesh
Miscellaneous	AddFrame
	AddMeshBuilder
	ReserveSpace
	Save
	Scale
	SetMaterial
	Translate
Normals	AddNormal
	GenerateNormals2
	SetNormal
Perspective	GetPerspective
	SetPerspective
Rendering quality	GetQuality
	SetQuality
Textures	GetTextureCoordinates
	SetTexture
	SetTextureCoordinates

	SetTextureTopology
Vertices	AddVertex GetVertexColor GetVertexCount GetVertices SetVertex SetVertexColor SetVertexColorRGB

The **IDirect3DRMMeshBuilder2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

In addition, the **IDirect3DRMMeshBuilder2** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName
SetAppData
SetName

The Direct3DRMMeshBuilder2 object is obtained by calling the IDirect3DRM::CreateMeshBuilder method.

IDirect3DRMMeshBuilder2::AddFace

Adds a face to a Direct3DRMMeshBuilder2 object.

HRESULT AddFace(
    LPDIRECT3DRMFACE *lpD3DRMFace*
    );

Parameters

lpD3DRMFace
Address of the face being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Any one face can exist in only one mesh at a time.

IDirect3DRMMeshBuilder2::AddFaces

Adds faces to a Direct3DRMMeshBuilder2 object.

HRESULT AddFaces(
DWORD *dwVertexCount*,
D3DVECTOR * *lpD3DVertices*,
DWORD *normalCount*,
D3DVECTOR **lpNormals*,
DWORD **lpFaceData*,
LPDIRECT3DRMFACEARRAY* *lplpD3DRMFaceArray*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::AddFrame

Adds the contents of a frame to a Direct3DRMMeshBuilder2 object.

HRESULT AddFrame(
    LPDIRECT3DRMFRAME *lpD3DRMFrame*
    );

Parameters

lpD3DRMFrame
Address of the frame whose contents are being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The source frame is not modified or referenced by this operation.

IDirect3DRMMeshBuilder2::AddMesh

Adds a mesh to a Direct3DRMMeshBuilder2 object.

HRESULT AddMesh(
    LPDIRECT3DRMMESH *lpD3DRMMesh*
    );

Parameters

lpD3DRMMesh
Address of the mesh being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::AddMeshBuilder

Adds the contents of a Direct3DRMMeshBuilder object to a Direct3DRMMeshBuilder2 object.

HRESULT AddMeshBuilder(
    LPDIRECT3DRMMESHBUILDER *lpD3DRMMeshBuild*
    );

Parameters

lpD3DRMMeshBuild
Address of the Direct3DRMMeshBuilder object whose contents are being added.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The source Direct3DRMMeshBuilder object is not modified or referenced by this operation.

IDirect3DRMMeshBuilder2::AddNormal

Adds a normal to a Direct3DRMMeshBuilder2 object.

int AddNormal(
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

x, y, and *z*
The x, y, and z components of the direction of the new normal.

Return Values

Returns the index of the normal.

IDirect3DRMMeshBuilder2::AddVertex

Adds a vertex to a Direct3DRMMeshBuilder2 object.

int AddVertex(
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

x, y, and *z*
The x, y, and z components of the position of the new vertex.

Return Values

Returns the index of the vertex.

IDirect3DRMMeshBuilder2::CreateFace

Creates a new face with no vertices and adds it to a Direct3DRMMeshBuilder2 object.

HRESULT CreateFace(
    LPDIRECT3DRMFACE* *lplpD3DRMFace*
    );

Parameters

lplpD3DRMFace

Address of a pointer to an IDirect3DRMFace interface that will be filled with a pointer to the face that was created.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::CreateMesh

Creates a new mesh from a Direct3DRMMeshBuilder2 object.

HRESULT CreateMesh(
    LPDIRECT3DRMMESH* *lplpD3DRMMesh*
    );

Parameters

lplpD3DRMMesh
Address that will be filled with a pointer to an IDirect3DRMMesh interface.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::GenerateNormals2

Processes the Direct3DRMMeshBuilder2 object and generates normals for each vertex in a mesh by averaging the face normals for each face that shares the vertex. New normals are generated if the faces sharing a vertex have an angle between them greater than the crease angle.

HRESULT GenerateNormals2(
D3DVALUE *dvCreaseAngle*,
DWORD *dwFlags*
);

Parameters

dvCreaseAngle

The least angle in radians that faces can have between them and have a new normal generated.

dwFlags

One of the following values.

D3DRMGENERATENORMALS_PRECOMPACT (*dwFlags* = 1) – Specifies that the algorithm should attempt to compact mesh vertices before it generates normals. See comments below.

D3DRMGENERATENORMALS_USECREASEANGLE (*dwFlags* = 2) – Specifies that the *dvCreaseAngle* parameter should be used. Otherwise, the crease angle is ignored.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

If you specify the D3DRMGENERATENORMALS_PRECOMPACT flag, the precompact pass searches all the vertices in the mesh and merges any that are the same. This is a good way of compacting a mesh that has been loaded. Some meshes have multiple vertices if they have multiple normals at that vertex. This is not necessary in Direct3D Retained Mode. Specifying this flag is the way to get rid of the multiple vertices.

After compacting, the normals are generated. The edges between faces are examined, and if the angle the faces make at the edge is less than the crease angle, the face normals are averaged to generate the vertex normal. If the angle is greater than the crease angle, a new normal is generated. Note that a new vertex is not generated.

IDirect3DRMMeshBuilder2::GetBox

Retrieves the bounding box containing a Direct3DRMMeshBuilder2 object. The bounding box gives the minimum and maximum model coordinates in each dimension.

HRESULT GetBox(
D3DRMBOX **lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::GetColorSource

Retrieves the color source of a Direct3DRMMeshBuilder2 object. The color source can be either a face or a vertex.

D3DRMCOLORSOURCE GetColorSource( );

Return Values

Returns a member of the D3DRMCOLORSOURCE enumerated type.

See Also

IDirect3DRMMeshBuilder2::SetColorSource

IDirect3DRMMeshBuilder2::GetFace

Retrieves a single face of a Direct3DRMMeshBuilder2 object.

HRESULT GetFace(
DWORD *dwIndex*,
LPDIRECT3DRMFACE* *lplpD3DRMFace*
);

Parameters

dwIndex
The index of the mesh face to be retrieved. The face must already be part of a Direct3DRMMeshBuilder2 object.

lplpD3DRMFace
Address of a pointer to an IDirect3DRMFace interface that is filled with an address of the face.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::GetFaceCount

Retrieves the number of faces in a Direct3DRMMeshBuilder2 object.

int GetFaceCount( );

Return Values

Returns the number of faces.

IDirect3DRMMeshBuilder2::GetFaces

Retrieves the faces of a Direct3DRMMeshBuilder2 object.

HRESULT GetFaces(
    LPDIRECT3DRMFACEARRAY* *lplpD3DRMFaceArray*
    );

Parameters

lplpD3DRMFaceArray
Address of a pointer to an IDirect3DRMFaceArray interface that is filled with an address of the faces.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::GetPerspective

Determines whether perspective correction is on for a Direct3DRMMeshBuilder2 object.

BOOL GetPerspective( );

Return Values

Returns TRUE if perspective correction is on, or FALSE otherwise.

IDirect3DRMMeshBuilder2::GetQuality

Retrieves the rendering quality of a Direct3DRMMeshBuilder2 object.

D3DRMRENDERQUALITY GetQuality( );

Return Values

Returns a member of the D3DRMRENDERQUALITY enumerated type that specifies the rendering quality of the mesh.

See Also

IDirect3DRMMeshBuilder2::SetQuality

IDirect3DRMMeshBuilder2::GetTextureCoordinates

Retrieves the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT GetTextureCoordinates(
DWORD *index*,
D3DVALUE **lpU*,
D3DVALUE **lpV*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder2::SetTextureCoordinates

IDirect3DRMMeshBuilder2::GetVertexColor

Retrieves the color of a specified vertex in a Direct3DRMMeshBuilder2 object.

D3DCOLOR GetVertexColor(
    DWORD *index*
    );

Parameters

index
Index of the vertex.

Return Values

Returns the color.

See Also

IDirect3DRMMeshBuilder2::SetVertexColor

IDirect3DRMMeshBuilder2::GetVertexCount

Retrieves the number of vertices in a Direct3DRMMeshBuilder2 object.

int GetVertexCount( );

Return Values

Returns the number of vertices.

IDirect3DRMMeshBuilder2::GetVertices

Retrieves the vertices, normals, and face data for a Direct3DRMMeshBuilder2 object.

HRESULT GetVertices(
DWORD **vcount*,
D3DVECTOR **vertices*,
DWORD **ncount*,
D3DVECTOR **normals*,
DWORD **face_data_size*,
DWORD **face_data*
);

Parameters

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 Direct3DRMMeshBuilder2 object. If this parameter is NULL, the method returns the number of vertices in the *vcount* parameter.

ncount
Address of a variable that will contain the number of normals.

normals
Address of an array of **D3DVECTOR** structures that will contain the normals for the Direct3DRMMeshBuilder2 object. If this parameter is NULL, the method returns the number of normals in the *ncount* parameter.

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 Direct3DRMMeshBuilder2 object. This data is in the same format as specified in the IDirect3DRMMeshBuilder2::AddFaces 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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::Load

Loads a Direct3DRMMeshBuilder2 object.

HRESULT Load(
LPVOID *lpvObjSource*,
LPVOID *lpvObjID*,
D3DRMLOADOPTIONS *d3drmLOFlags*,
D3DRMLOADTEXTURECALLBACK *d3drmLoadTextureProc*,
LPVOID *lpvArg*
);

Parameters

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 that require special formatting. This parameter can be NULL.

lpvArg
Address of application-defined data passed to the D3DRMLOADTEXTURECALLBACK callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

By default, this method loads the first mesh from the source specified in the *lpvObjSource* parameter.

IDirect3DRMMeshBuilder2::ReserveSpace

Reserves space within a Direct3DRMMeshBuilder2 object for the specified number of vertices, normals, and faces. This allows the system to use memory more efficiently.

HRESULT ReserveSpace(
DWORD *vertexCount*,
DWORD *normalCount*,
DWORD *faceCount*
);

Parameters

vertexCount, normalCount, and *faceCount*
Number of vertices, normals, and faces to allocate space for.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::Save

Saves a Direct3DRMMeshBuilder2 object.

HRESULT Save(
const char * *lpFilename*,
D3DRMXOFFORMAT *d3drmXOFFormat*,
D3DRMSAVEOPTIONS *d3drmSOContents*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::Scale

Scales a Direct3DRMMeshBuilder2 object by the given scaling factors, parallel to the x-, y-, and z-axes in model coordinates.

HRESULT Scale(
D3DVALUE *sx*,
D3DVALUE *sy*,
D3DVALUE *sz*
);

Parameters

sx, sy, and *sz*
Scaling factors that are applied along the x-, y-, and z-axes.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetColor

Sets all the faces of a Direct3DRMMeshBuilder2 object to a given color.

HRESULT SetColor(
    D3DCOLOR *color*
    );

Parameters

color
Color of the faces.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetColorRGB

Sets all the faces of a Direct3DRMMeshBuilder2 object to a given color.

HRESULT SetColorRGB(
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

red, green, and *blue*
Red, green, and blue components of the color.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetColorSource

Sets the color source of a Direct3DRMMeshBuilder2 object.

HRESULT SetColorSource(
D3DRMCOLORSOURCE *source*
);

Parameters

source
Member of the D3DRMCOLORSOURCE enumerated type that specifies the new color source to use.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder2::GetColorSource

IDirect3DRMMeshBuilder2::SetMaterial

Sets the material of all the faces of a Direct3DRMMeshBuilder2 object.

HRESULT SetMaterial(
    LPDIRECT3DRMMATERIAL *lpIDirect3DRMmaterial*
    );

Parameters

lpIDirect3DRMmaterial
Address of IDirect3DRMMaterial interface for the Direct3DRMMeshBuilder2 object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetNormal

Sets the normal vector of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT SetNormal(
DWORD *index*,
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetPerspective

Enables or disables perspective-correct texture-mapping for a Direct3DRMMeshBuilder2 object.

HRESULT SetPerspective(
BOOL *perspective*
);

Parameters

perspective
Specify TRUE if the mesh should be texture-mapped with perspective correction, or FALSE otherwise.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetQuality

Sets the rendering quality of a Direct3DRMMeshBuilder2 object.

HRESULT SetQuality(
D3DRMRENDERQUALITY *quality*
);

Parameters

quality

Member of the D3DRMRENDERQUALITY enumerated type that specifies the new rendering quality to use.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

An object's quality has three components: shade mode (flat or gouraud, phong is not yet implemented and will default to gouraud shading), lighting type (on or off), and fill mode (point, wireframe or solid).

You can set the quality of a device with IDirect3DRMDevice::SetQuality. By default it is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMMeshBuilder2 object with the **SetQuality** method. By default, a Direct3DRMMeshBuilder2 object's quality is D3DRMRENDER_GOURAUD (gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

See Also

IDirect3DRMMeshBuilder2::GetQuality

IDirect3DRMMeshBuilder2::SetTexture

Sets the texture of all the faces of a Direct3DRMMeshBuilder2 object.

HRESULT SetTexture(
    LPDIRECT3DRMTEXTURE *lpD3DRMTexture*
    );

Parameters

lpD3DRMTexture
Address of the required Direct3DRMTexture object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetTextureCoordinates

Sets the texture coordinates of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT SetTextureCoordinates(
DWORD *index*,
D3DVALUE *u*,
D3DVALUE *v*
);

Parameters

index
Index of the vertex to be set.

u and *v*
Texture coordinates to assign to the specified mesh vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder2::GetTextureCoordinates

IDirect3DRMMeshBuilder2::SetTextureTopology

Sets the texture topology of a Direct3DRMMeshBuilder2 object.

HRESULT SetTextureTopology(
BOOL *cylU*,
BOOL *cylV*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetVertex

Sets the position of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT SetVertex(
DWORD *index*,
D3DVALUE *x*,
D3DVALUE *y*,
D3DVALUE *z*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::SetVertexColor

Sets the color of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT SetVertexColor(
DWORD *index*,
D3DCOLOR *color*
);

Parameters

index
Index of the vertex to be set.

color
Color to assign to the specified vertex.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMMeshBuilder2::GetVertexColor

IDirect3DRMMeshBuilder2::SetVertexColorRGB

Sets the color of a specified vertex in a Direct3DRMMeshBuilder2 object.

HRESULT SetVertexColorRGB(
DWORD *index*,
D3DVALUE *red*,
D3DVALUE *green*,
D3DVALUE *blue*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMMeshBuilder2::Translate

Adds the specified offsets to the vertex positions of a Direct3DRMMeshBuilder2 object.

HRESULT Translate(
D3DVALUE *tx*,
D3DVALUE *ty*,
D3DVALUE *tz*
);

Parameters

tx, ty, and *tz*
Offsets that are added to the x-, y-, and z-coordinates respectively of each vertex position.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObject

Applications use the methods of the **IDirect3DRMObject** interface to work with the object superclass of Direct3DRM objects. This section is a reference to the methods of this interface. For a conceptual overview, see Direct3DRMObject.

The methods of the **IDirect3DRMObject** interface can be organized into the following groups:

Application-specific data	GetAppData SetAppData
Cloning	Clone
Naming	GetClassName GetName SetName
Notifications	AddDestroyCallback DeleteDestroyCallback

The **IDirect3DRMObject** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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

Registers a function to be called when an object is destroyed.

HRESULT AddDestroyCallback(
D3DRMOBJECTCALLBACK *lpCallback*,
LPVOID *lpArg*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObject::Clone

Creates a copy of an object.

HRESULT Clone(
LPUNKNOWN *pUnkOuter*,
REFIID *riid*,
LPVOID **ppvObj*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObject::DeleteDestroyCallback

Removes a function previously registered with the IDirect3DRMObject::AddDestroyCallback method.

HRESULT DeleteDestroyCallback(
    D3DRMOBJECTCALLBACK *d3drmObjProc*,
    LPVOID *lpArg*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObject::GetAppData

Retrieves the 32 bits of application-specific data in the object. The default value is 0.

DWORD GetAppData( );

Return Values

Returns the data value defined by the application.

See Also

IDirect3DRMObject::SetAppData

IDirect3DRMObject::GetClassName

Retrieves the name of the object's class.

HRESULT GetClassName(
    LPDWORD *lpdwSize*,
    LPSTR *lpName*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMObject::GetName

Retrieves the object's name.

HRESULT GetName(
    LPDWORD *lpdwSize*,
    LPSTR *lpName*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMObject::SetName

IDirect3DRMObject::SetAppData

Sets the 32 bits of application-specific data in the object.

HRESULT SetAppData(
    DWORD *ulData*
    );

Parameters

ulData
User-defined data to be stored with the object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMObject::GetAppData

IDirect3DRMObject::SetName

Sets the object's name.

HRESULT SetName(
    const char * *lpName*
    );

Parameters

lpName
User-defined data to be the name for the object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMObject::GetName

IDirect3DRMProgressiveMesh

The Progressive Mesh is a form of mesh that allows for progressive refinement. Conceptually, it consists of a base mesh representation and a number of "vertex split" records. A mesh can be stored as a much coarser mesh together with records of how to incrementally refine the mesh. This allows for a generalized level of detail to be set on the mesh as well as progressive download of the mesh from a remote source.

The progressive mesh is a visual in Direct3D Retained Mode and may be used in the standard ways that visuals are used. That is, a progressive mesh can be added to frame hierarchies, multiply instanced and picked.

The **Direct3DRMProgressiveMesh** object is created by calling the IDirect3DRM2::CreateProgressiveMesh method. After creation, the object can be added to a hierarchy, but will not render until at least the base mesh is available.

For a conceptual overview, see IDirect3DRMProgressiveMesh.

In addition to the standard *IUnknown* and IDirect3DRMObject methods, this API contains the following members:

Creating and Copying Meshes	Clone
	CreateMesh
	Duplicate
	GetBox
Loading	Abort
	GetLoadStatus
	Load
Setting Quality	SetQuality
	GetQuality
Managing Details	GetDetail
	GetFaceDetail
	GetFaceDetailRange
	GetVertexDetail
	GetVertexDetailRange
	SetDetail
	SetFaceDetail
	SetMinRenderDetail
	SetVertexDetail
Registering Events	RegisterEvents

The **IDirect3DRMProgressiveMesh** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

In addition, the **IDirect3DRMProgressiveMesh** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback
Clone
DeleteDestroyCallback
GetAppData
GetClassName
GetName

SetAppData
SetName

IDirect3DRMProgressiveMesh::Abort

Terminates the currently active download.

HRESULT Abort(
DWORD *dwFlags*
)

Parameters

dwFlags
Must be set to zero.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS

Remarks

If the base mesh has been downloaded before this method is called, the effect is as if the progressive mesh has loaded, the vertex splits are in a valid state, and the progressive mesh is renderable. The other progressive mesh methods will work. If the base mesh has not been downloaded before the **IDirect3DRMProgressiveMesh::Abort** call and you have added the progressive mesh to a scene, the **Render** will succeed but the progressive mesh will not be rendered. Also, if the base mesh has not been downloaded, when you try to use the progressive mesh (to create a mesh or clone, for example) the call will return D3DRMERR_NOTENOUGHDATA.

If there are any outstanding events, they are signaled.

IDirect3DRMProgressiveMesh::Clone

Creates a copy of the currently loaded **Direct3DRMProgressiveMesh** object.

HRESULT Clone(
LPDIRECT3DRMPROGRESSIVEMESH* *lplpD3DRMPMesh*
)

Parameters

lplpD3DRMPMesh
Address of a **Direct3DRMProgressiveMesh** pointer that will be filled in with a pointer to the newly generated **Direct3DRMProgressiveMesh** object.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_NOTENOUGHDATA

Remarks

The progressive mesh being cloned must have at least its base mesh loaded. If you call this method on a progressive mesh that is currently being asynchronously loaded, the cloned mesh only has as much detail as the loading progressive mesh had at the time it was cloned. Any vertex splits loaded after the duplication are not available to the cloned mesh.

This method does not try to share any of the progressive mesh's internal data, whereas the IDirect3DRMProgressiveMesh::Duplicate does.

See Also

IDirect3DRMProgressiveMesh::Duplicate

IDirect3DRMProgressiveMesh::CreateMesh

Builds a mesh from the current level of detail.

HRESULT CreateMesh(
LPDIRECT3DRMMESH* *lplpD3DRMMesh*
)

Parameters

lplpD3DRMMesh
Address of a **Direct3DRMMesh** pointer that will be filled in with a pointer to the newly generated **Direct3DRMMesh** object.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_NOTENOUGHDATA

Remarks

If the application has requested a level of detail that is not yet available, or the base mesh is not yet available, this method returns the error D3DRMERR_NOTENOUGHDATA.

IDirect3DRMProgressiveMesh::Duplicate

Creates a copy of the **Direct3DRMProgressiveMesh** object. The copy shares all geometry and face data with the original, but has a detail level that can be set independently of the original. This enables the same mesh data to be used in different parts of the hierarchy but with different levels of detail. In essence, you almost have two instances of a progressive mesh within the frame hierarchy.

HRESULT Duplicate(
LPDIRECT3DRMPROGRESSIVEMESH* *lplpD3DRMPMesh*
)

Parameters

lplpD3DRMPMesh
Address of a **Direct3DRMProgressiveMesh** pointer that will be filled with a pointer to the newly generated **Direct3DRMProgressiveMesh** object.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_NOTENOUGHDATA

Remarks

The progressive mesh being duplicated must have at least its base mesh loaded. If you call this method on a progressive mesh that is currently being asynchronously loaded, the duplicated mesh only has as much detail as the loading progressive mesh had at the time it was duplicated. Any vertex splits loaded after the duplication are not available to the duplicated mesh.

A progressive mesh has a set of data representing the base mesh, and a set of data representing the vertex splits. The base mesh data and data describing the current state of the progressive mesh and the current level of detail isn't shared between meshes that are duplicated, but the vertex splits are. This method creates a new instance of the Progressive Mesh which shares all geometry and face data with the original, but has a detail level that can be set independently of the original.

IDirect3DRMProgressiveMesh::GetBox

Retrieves the bounding box containing a Direct3DRMProgressiveMesh object. The bounding box gives the minimum and maximum coordinates relative to a child frame, in each dimension.

HRESULT GetBox(
D3DRMBOX * *lpD3DRMBox*
);

Parameters

lpD3DRMBox
Address of a D3DRMBOX structure that will be filled with the bounding box coordinates.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMProgressiveMesh::GetDetail

Returns the current detail level of the progressive mesh normalized between 0.0 and 1.0.

HRESULT GetDetail(
LPD3DVALUE *lpdvVal*
)

Parameters

lpdvVal
Address of a **D3DVALUE** that will be filled with the current detail level of the progressive mesh.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING

Remarks

If the base mesh has not yet downloaded then this method will return D3DRMERR_PENDING. If a requested level of detail has been set, the return value will increase on each subsequent call until the requested level has been met. If no level has been requested, then the detail will increase until all vertex splits have been downloaded.

The normalized value 0.0 represents the minimum number of vertices (the number of vertices in the base mesh), and the normalized value 1.0 represents the maximum number of vertices.

IDirect3DRMProgressiveMesh::GetFaceDetail

Retrieves the number of faces in the progressive mesh.

HRESULT GetFaceDetail(
LPDWORD *lpdwCount*
)

Parameters

lpdwCount
Address of a **DWORD** that will be filled in with the number of faces in the progressive mesh.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_PENDING
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS

Remarks

If the number of faces is not available, this method returns D3DRMERR_PENDING.

IDirect3DRMProgressiveMesh::GetFaceDetailRange

Retrieves the minimum and maximum face count available in the progressive mesh.

HRESULT GetFaceDetailRange(
LPDWORD *lpdwMinFaces*,
LPDWORD *lpdwMaxFaces*
)

Parameters

lpdwMinFaces
Address of a **DWORD** that will be filled in with the minimum number of faces.
lpdwMaxFaces
Address of a **DWORD** that will be filled in with the maximum number of faces.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING

Remarks

If the face count is not available, this method returns D3DRMERR_PENDING.

IDirect3DRMProgressiveMesh::GetLoadStatus

Allows the application to inquire about the current status of the load.

HRESULT GetLoadStatus(
LPD3DRMPMESHLOADSTATUS *lpStatus*
)

Parameters

lpStatus

Address of a D3DRMPMESHLOADSTATUS structure defined as follows:

```
typedef struct _D3DRMPMESHLOADSTATUS
{
  DWORD dwSize;      // Size of this structure
  DWORD dwPMeshSize;   // Total size (bytes)
  DWORD dwBaseMeshSize;      // Size of base mesh (bytes)
  DWORD dwBytesLoaded;   // Total number of bytes loaded
  DWORD dwVerticesLoaded;          // Number of vertices loaded
  DWORD dwFacesLoaded;   // Number of faces loaded
  DWORD dwFlags;
}
D3DRMPMESHLOADSTATUS;
typedef D3DRMPMESHLOADSTATUS *LPD3DRMPMESHLOADSTATUS;
```

The *dwFlags* member can take the following values:

D3DRMPMESHSTATUS_VALID – The progressive mesh object contains valid data.

D3DRMPMESHSTATUS_INTERRUPTED – The download was interrupted either because the application called Abort or because the connection was lost.

D3DRMPMESHSTATUS_BASEMESH – The base mesh has been downloaded.

D3DRMPMESHSTATUS_COMPLETE – All data has been downloaded.

D3DRMPMESHSTATUS_RENDERABLE – It is now possible to render the mesh.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS

Remarks

If the mesh is renderable (the base mesh has been downloaded and the data is not corrupt) then the *dwFlags* member contains D3DRMPMESHSTATUS_RENDERABLE. If the download was interrupted, the *dwFlags* member will contain D3DRMPMESHSTATUS_INTERRUPTED.

IDirect3DRMProgressiveMesh::GetQuality

Retrieves a member of the D3DRMRENDERQUALITY enumerated type that specifies the rendering quality of the progressive mesh.

HRESULT GetQuality(
LPD3DRMRENDERQUALITY *lpQuality*
);

Parameters

lpQuality
Pointer to the D3DRMRENDERQUALITY member if successful that specifies rendering quality.

Return Values

Returns D3DRM_OK if successful, or one of the following errors:

D3DRMERR_BADOBJECT - the progressive mesh is invalid.
D3DRMERR_BADVALUE - the pointer to the D3DRMRENDERQUALITY member is invalid.

See Also

IDirect3DRMProgressiveMesh::SetQuality

IDirect3DRMProgressiveMesh::GetVertexDetail

Retrieves the number of vertices in the progressive mesh.

HRESULT GetVertexDetail(
LPDWORD *lpdwCount*
)

Parameters

lpdwCount
Address of a **DWORD** that will be filled in with the number of vertices in the progressive mesh.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING

Remarks

If the number of vertices is not available, this method returns D3DRMERR_PENDING.

IDirect3DRMProgressiveMesh::GetVertexDetailRange

Retrieves the minimum and maximum vertex count available in the progressive mesh.

HRESULT GetVertexDetailRange(
LPDWORD *lpdwMinVertices*,
LPDWORD *lpdwMaxVertices*
)

Parameters

lpdwMinVertices
Address of a **DWORD** that will be filled in with the minimum number of vertices.
lpdwMaxVertices
Address of a **DWORD** that will be filled in with the maximum number of vertices.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING

Remarks

If the vertex count information is not available, this method returns D3DRMERR_PENDING.

IDirect3DRMProgressiveMesh::Load

Loads the progressive mesh object from memory, a file, a resource, or a URL. Loading can be done synchronously or asynchronously.

HRESULT Load(
LPVOID *lpSource*,
LPVOID *lpObjID*,
D3DRMLOADOPTIONS *dloLoadflags*,
D3DRMLOADTEXTURECALLBACK *lpCallback*,
LPVOID *lpArg*
)

Parameters

lpSource

Address of the 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 *dloLoadflags* parameter.

lpObjID

Address of the ID of the DirectX file record that is a Progressive Mesh. This can either be a string or a UUID (determined by *dloLoadflags*). If *lpObjID* is NULL, then *dloLoadflags* must be D3DRMLOAD_FIRST.

dloLoadflags

Value of the D3DRMLOADOPTIONS type describing how the load is to be performed. One flag from each of the following two groups must be included:

The following values determine which object in the DirectX file is loaded:

D3DRMLOAD_BYNAME	The *lpObjID* parameter is interpreted as a string.
D3DRMLOAD_BYGUID	The *lpObjID* parameter is interpreted as a UUID.
D3DRMLOAD_FIRST	The first progressive mesh found is loaded.

The following flags determine the source of the DirectX file:

D3DRMLOAD_FROMFILE	The *lpSource* parameter is interpreted as a string representing a local file name.
D3DRMLOAD_FROMRESOURCE	The *lpSource* parameter is interpreted as a pointer to a D3DRMLOADRESOURCE structure.
D3DRMLOAD_FROMMEMORY	The *lpSource* parameter is interpreted as a pointer to a D3DRMLOADMEMORY structure.
D3DRMLOAD_FROMURL	The *lpSource* parameter is interpreted as a URL.

In addition, you can specify whether the download is synchronous or asynchronous. By default, loading is synchronous and the **Load** call will not return until all data has been loaded or an error occurs. To specify asynchronous loading, include the following flag:

D3DRMLOAD_ASYNCHRONOUS – The **Load** call will return immediately. It is up to the application to use events with IDirect3DRMProgressiveMesh::RegisterEvents and IDirect3DRMProgressiveMesh::GetLoadStatus to find out how the load is progressing.

lpCallback

Address of a D3DRMLOADTEXTURECALLBACK callback function that will be called to load any texture encountered. The callback will be called with the texture name as encountered by the loader and the user-defined *lpArg* parameter. A new thread is not spawned for the callback. If you want the application to download a texture progressively, it must spawn a thread and return with a LPDIRECT3DRMTEXTURE as normal.

lpArg

Address of user-defined data passed to the D3DRMLOADTEXTURECALLBACK callback function.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_BADPMDATA
D3DRMERR_BADFILE
D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS

Remarks

The progressive mesh is not useful unless it has been initialized. If asynchronous download is specified, the API returns immediately and a separate thread is spawned to perform the download.

This method, IDirect3DRMProgressiveMesh::Clone, and IDirect3DRMProgressiveMesh::Duplicate all initialize a progressive mesh. You can only initialize an object once. Therefore, you cannot clone or duplicate a progressive mesh and then try to load into the cloned or duplicated mesh, nor can you load into a previously loaded mesh.

IDirect3DRMProgressiveMesh::RegisterEvents

Allows the application to register events with the progressive mesh object that will be signaled when the appropriate conditions are met.

HRESULT RegisterEvents(
HANDLE *hEvent*,
DWORD *dwFlags*,
DWORD *dwReserved*
)

Parameters

hEvent
Event to be signaled when the required condition is met.

dwFlags
Can be one of the following flags:

D3DRMPMESHEVENT_BASEMESH	The event is signaled when the base mesh has been downloaded.
D3DRMPMESHEVENT_COMPLETE	The event is signaled when all data has been downloaded.

dwReserved
Must be zero.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS

Remarks

This method can be used to monitor the progress of loads. Events will also be signaled if an error occurs, so your application should always call the IDirect3DRMProgressiveMesh::GetLoadStatus method after being signaled.

IDirect3DRMProgressiveMesh::SetDetail

Sets a requested level of detail normalized between 0.0 and 1.0.

HRESULT SetDetail(
D3DVALUE *dvVal*
)

Parameters

dvVal
The requested level of detail.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_BADPMDATA
D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING

Remarks

If not enough detail has been downloaded yet (but will be available), the request is acknowledged and D3DRMERR_PENDING is returned. This error is informational and simply indicates that the requested level will be set as soon as enough detail is available.

The normalized value 0.0 represents the minimum number of vertices (the number of vertices in the base mesh), and the normalized value 1.0 represents the maximum number of vertices.

IDirect3DRMProgressiveMesh::SetFaceDetail

Sets the requested level of face detail.

HRESULT SetFaceDetail(
DWORD *dwCount*
)

Parameters

dwCount
The number of faces requested.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_BADPMDATA
D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING
D3DRMERR_REQUESTTOOLARGE

Remarks

Sometimes it is not possible to set the progressive mesh to the number of faces requested, though it will always be within 1 of the requested value. This is because a vertex split can add 1 or 2 faces. For example, if you call **SetFaceDetail**(20), the progressive mesh may only be able to set the face detail to 19 or 21. You can always get the actual number of faces in the progressive mesh by calling the IDirect3DRMProgressiveMesh::GetFaceDetail method.

If not enough detail has been downloaded yet (but will be available), the request is be acknowledged and D3DRMERR_PENDING is returned. This error is informational and simply indicates that the requested level will be set as soon as enough detail is available. If the detail requested is greater that the detail available in the progressive mesh then D3DRMERR_REQUESTTOOLARGE is returned.

See Also

IDirect3DRMProgressiveMesh::GetFaceDetail

IDirect3DRMProgressiveMesh::SetMinRenderDetail

Sets the minimum level of detail that will be rendered during a load from 0.0 (minimum detail) to 1.0 (maximum detail). Normally, the progressive mesh will be rendered once the base mesh is available (and the mesh is in the scene graph).

HRESULT SetMinRenderDetail(
D3DVALUE *dvCount*
)

Parameters

dvCount
The requested minimum detail.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING
D3DRMERR_REQUESTTOOLARGE
D3DRMERR_REQUESTTOOSMALL

Remarks

This API sets the minimum number of faces/vertices that will be rendered during the load (larger than the base mesh).

Any subsequent IDirect3DRMProgressiveMesh::SetDetail, IDirect3DRMProgressiveMesh::SetFaceDetail, or IDirect3DRMProgressiveMesh::SetVertexDetail calls will override this.

IDirect3DRMProgressiveMesh::SetQuality

Sets the rendering quality of a Direct3DRMProgressiveMesh object.

HRESULT SetQuality(
D3DRMRENDERQUALITY *quality*
);

Parameters

quality

Member of the D3DRMRENDERQUALITY enumerated type that specifies the new rendering quality to use.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

An object's quality has three components: shade mode (flat or gouraud, phong is not yet implemented and will default to gouraud shading), lighting type (on or off), and fill mode (point, wireframe or solid).

You can set the quality of a device with IDirect3DRMDevice::SetQuality. By default it is D3DRMRENDER_FLAT (flat shading, lights on, and solid fill).

You can set the quality of a Direct3DRMProgressiveMesh object with the **SetQuality** method. By default, a Direct3DRMProgressiveMesh object's quality is D3DRMRENDER_GOURAUD (gouraud shading, lights on, and solid fill).

DirectX Retained Mode renders an object at the lowest quality setting based on the device and object's current setting for each individual component. For example, if the object's current quality setting is D3DRMRENDER_GOURAUD, and the device is D3DRMRENDER_FLAT then the object will be rendered with flat shading, solid fill and lights on.

If the object's current quality setting is D3DRMSHADE_GOURAUD|D3DRMLIGHT_OFF| D3DRMFILL_WIREFRAME and the device's quality setting is D3DRMSHADE_FLAT| D3DRMLIGHT_ON|D3DRMFILL_POINT, then the object will be rendered with flat shading, lights off and point fill mode.

These rules apply to Direct3DRMMeshBuilder objects, Direct3DRMMeshBuilder2 objects, and Direct3DRMProgressiveMesh objects. However, Direct3DRMMesh objects do not follow these rules. Mesh objects ignore the device's quality settings and use the group quality setting (which defaults to D3DRMRENDER_GOURAUD).

See Also

IDirect3DRMProgressiveMesh::GetQuality

IDirect3DRMProgressiveMesh::SetVertexDetail

Sets the requested level of vertex detail.

HRESULT SetVertexDetail(
DWORD *dwCount*
)

Parameters

dwCount
The number of vertices requested.

Return Values

Returns DD_OK if successful, or one of the following errors:

D3DRMERR_BADPMDATA
D3DRMERR_CONNECTIONLOST
D3DRMERR_INVALIDDATA
D3DRMERR_INVALIDOBJECT
D3DRMERR_INVALIDPARAMS
D3DRMERR_PENDING
D3DRMERR_REQUESTTOOLARGE

Remarks

If not enough detail has been downloaded yet (but will be available), the request will be acknowledged and D3DRMERR_PENDING is returned. This error is informational and simply indicates that the requested level will be set as soon as enough detail is available. If the detail requested is greater that the detail available in the progressive mesh then D3DRMERR_REQUESTTOOLARGE is returned.

See Also

IDirect3DRMProgressiveMesh::GetVertexDetail

IDirect3DRMShadow

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 section is a reference to the methods of the **IDirect3DRMShadow** interface. For a conceptual overview, see IDirect3DRMShadow Interface.

The **IDirect3DRMShadow** interface supports the Init method.

The **IDirect3DRMShadow** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::Init

Initializes a Direct3DRMShadow object.

HRESULT Init(
LPDIRECT3DRMVISUAL *lpD3DRMVisual*,
LPDIRECT3DRMLIGHT *lpD3DRMLight*,
D3DVALUE *px*,
D3DVALUE *py*,
D3DVALUE *pz*,
D3DVALUE *nx*,
D3DVALUE *ny*,
D3DVALUE *nz*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMTexture

Applications use the methods of the **IDirect3DRMTexture** interface to work with textures, which are rectangular arrays of pixels. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMTexture and IDirect3DRMTexture2 Interfaces.

To avoid unnecessary delays when creating textures, hold onto textures you want to use again, instead of creating them each time they're needed. For optimal performance, use a texture surface format that is supported by the device you are using. This will avoid a costly format conversion when the texture is created and any time it changes.

See the remarks in IDirect3DRM::LoadTexture for an example showing how to keep a reference to textures loaded in a texture callback through **IDirect3DRM::LoadTexture**.

The methods of the **IDirect3DRMTexture** interface 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
Renderer notification	Changed
Shading	GetShades SetShades

The **IDirect3DRMTexture** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::Changed

Informs the renderer that the application has changed the pixels or the palette of a texture.

HRESULT Changed(
    BOOL *bPixels*,
    BOOL *bPalette*
    );

Parameters

bPixels
    If this parameter is TRUE, the pixels have changed.

bPalette
    If this parameter is TRUE, the palette has changed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMTexture::GetColors

Retrieves the maximum number of colors used for rendering a texture.

DWORD GetColors( );

Return Values

Returns the number of colors.

Remarks

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.

See Also

IDirect3DRMTexture::SetColors

IDirect3DRMTexture::GetDecalOrigin

Retrieves the current origin of the decal.

HRESULT GetDecalOrigin(
LONG * *lplX*,
LONG * *lplY*
);

Parameters

lplX and *lplY*
Addresses of variables that will be filled with the origin of the decal when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::SetDecalOrigin

IDirect3DRMTexture::GetDecalScale

Retrieves the scaling property of the given decal.

DWORD GetDecalScale( );

Return Values

Returns the scaling property if successful, or -1 otherwise.

See Also

IDirect3DRMTexture::SetDecalScale

IDirect3DRMTexture::GetDecalSize

Retrieves the size of the decal.

HRESULT GetDecalSize(
D3DVALUE **lprvWidth*,
D3DVALUE **lprvHeight*
);

Parameters

lprvWidth and *lprvHeight*
Addresses of variables that will be filled with the width and height of the decal when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::SetDecalSize

IDirect3DRMTexture::GetDecalTransparency

Retrieves the transparency property of the decal.

BOOL GetDecalTransparency( );

Return Values

Returns TRUE if the decal has a transparent color, FALSE otherwise.

See Also

IDirect3DRMTexture::SetDecalTransparency

IDirect3DRMTexture::GetDecalTransparentColor

Retrieves the transparent color of the decal.

D3DCOLOR GetDecalTransparentColor( );

Return Values

Returns the value of the transparent color.

See Also

IDirect3DRMTexture::SetDecalTransparentColor

IDirect3DRMTexture::GetImage

Returns an address of the image that the texture was created with. Returns a NULL pointer if the current texture was created with a DirectDraw surface.

D3DRMIMAGE * GetImage( );

Return Values

Returns the address of the D3DRMIMAGE structure that the current texture was created with.

IDirect3DRMTexture::GetShades

Retrieves the number of shades used for each color in the texture when rendering.

DWORD GetShades( );

Return Values

Returns the number of shades.

See Also

IDirect3DRMTexture::SetShades

IDirect3DRMTexture::InitFromFile

Initializes a texture by using the information in a given file.

HRESULT InitFromFile(
    const char **filename*
    );

Parameters

filename
Address of a string specifying the file from which initialization information is drawn.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You must have created the texture to be initialized using the IDirect3DRM::CreateObject method.

See Also

IDirect3DRMTexture::InitFromResource, IDirect3DRMTexture::InitFromSurface

IDirect3DRMTexture::InitFromResource

Initializes a Direct3DRMTexture object from a specified resource.

HRESULT InitFromResource(
    HRSRC *rs*
    );

Parameters

rs
Handle of the specified resource.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::InitFromFile, IDirect3DRMTexture::InitFromSurface

IDirect3DRMTexture::InitFromSurface

Initializes a texture by using the data from a given DirectDraw surface.

HRESULT InitFromSurface(
    LPDIRECTDRAWSURFACE *lpDDS*
    );

Parameters

lpDDS
Address of a DirectDraw surface from which initialization information is drawn.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::InitFromFile, IDirect3DRMTexture::InitFromResource

IDirect3DRMTexture::SetColors

Sets the maximum number of colors used for rendering a texture. This method is required only in the ramp color model.

HRESULT SetColors(
    DWORD *ulColors*
    );

Parameters

ulColors
Number of colors. The default value is 8.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::GetColors

IDirect3DRMTexture::SetDecalOrigin

Sets the origin of the decal as an offset from the top left of the decal.

HRESULT SetDecalOrigin(
LONG *lX*,
LONG *lY*
);

Parameters

lX and *lY*
New origin, in decal coordinates, for the decal. The default origin is [0, 0].

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

This method is also used to add a decal origin key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture::GetDecalOrigin

IDirect3DRMTexture::SetDecalScale

Sets the scaling property for a decal.

HRESULT SetDecalScale(
    DWORD *dwScale*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::GetDecalScale

IDirect3DRMTexture::SetDecalSize

Sets the size of the decal to be used if the decal is being scaled according to its depth in the scene.

HRESULT SetDecalSize(
D3DVALUE *rvWidth*,
D3DVALUE *rvHeight*
);

Parameters

rvWidth and *rvHeight*
New width and height, in model coordinates, of the decal. The default size is [1, 1].

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a decal size key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture::GetDecalSize

IDirect3DRMTexture::SetDecalTransparency

Sets the transparency property of the decal.

HRESULT SetDecalTransparency(
    BOOL *bTransp*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::GetDecalTransparency

IDirect3DRMTexture::SetDecalTransparentColor

Sets the transparent color for a decal.

HRESULT SetDecalTransparentColor(
D3DCOLOR *rcTransp*
);

Parameters

rcTransp
New transparent color. The default transparent color is black.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a decal transparent color key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture::GetDecalTransparentColor

IDirect3DRMTexture::SetShades

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.

HRESULT SetShades(
    DWORD *ulShades*
    );

Parameters

ulShades
New number of shades. This value must be a power of 2. The default value is 16.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture::GetShades

IDirect3DRMTexture2

The **IDirect3DRMTexture2** interface is an extension of the IDirect3DRMTexture interface. IDirect3DRMTexture2::InitFromResource2 allows resources to be loaded from DLLs and executables other than the currently executing file. In addition, **IDirect3DRMTexture2** has two new methods. The IDirect3DRMTexture2::InitFromImage method creates a texture from an image in memory. This method is equivalent to the IDirect3DRM::CreateTexture method. The IDirect3DRMTexture2::GenerateMIPMap method generates a mipmap from a source image.

To avoid unnecessary delays when creating textures, hold onto textures you want to use again, instead of creating them each time they're needed. For optimal performance, use a texture surface format that is supported by the device you are using. This will avoid a costly format conversion when the texture is created and any time it changes.

See the remarks in IDirect3DRM2::LoadTexture for an example showing how to keep a reference to textures loaded in a texture callback through IDirect3DRM2::LoadTexture.

For a conceptual overview, see IDirect3DRMTexture and IDirect3DRMTexture2.

The methods of the **IDirect3DRMTexture2** interface 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 InitFromImage InitFromResource2 InitFromSurface
MIP map generation	GenerateMIPMap
Renderer notification	Changed
Shading	GetShades SetShades

The **IDirect3DRMTexture2** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

In addition, the **IDirect3DRMTexture2** interface inherits the following methods from the IDirect3DRMObject interface:

AddDestroyCallback

Clone

DeleteDestroyCallback

GetAppData

GetClassName

GetName

SetAppData

SetName

You can create an IDirect3DRMTexture2 object by calling IDirect3DRM2::CreateTexture or IDirect3DRM2::CreateTextureFromSurface.

IDirect3DRMTexture2::Changed

Informs the renderer that the application has changed the pixels or the palette of a Direct3DRMTexture2 object.

HRESULT Changed(
BOOL *bPixels*,
BOOL *bPalette*
);

Parameters

bPixels
If this parameter is TRUE, the pixels have changed.

bPalette
If this parameter is TRUE, the palette has changed.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMTexture2::GenerateMIPMap

Generates a mipmap from a single image source.

HRESULT GenerateMIPMap(
DWORD *dwFlags*
);

Parameters

dwFlags
Should be set to zero.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method can be called any time after a texture is created. It will generate a MIP map of the source image down to a resolution of 1×1 by using bi-linear filtering between levels. Once a MIP map has been generated, it will always be available and will be updated whenever IDirect3DRMTexture::Changed is called. When using MIP mapping, remember to change the texture quality to D3DRMTEXTURE_MIPNEAREST, D3DRMTEXTURE_MIPLINEAR, D3DRMTEXTURE_LINEARMIPNEAREST, or D3DRMTEXTURE_LINEARMIPLINEAR using IDirect3DRMDevice::SetTextureQuality. Extra MIP map levels will not be put into video memory for hardware devices unless the texture quality includes a MIP mapping type and the hardware device supports MIP mapping.

See Also

Mipmaps

IDirect3DRMTexture2::GetColors

Retrieves the maximum number of colors used for rendering a Direct3DRMTexture2 object.

DWORD GetColors( );

Return Values

Returns the number of colors.

Remarks

This method returns the number of colors that the Direct3DRMTexture2 object has been quantized to, not the number of colors in the image from which the Direct3DRMTexture2 object 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 IDirect3DRMTexture2::SetColors method explicitly to change the colors for the Direct3DRMTexture2 object.

See Also

IDirect3DRMTexture2::SetColors

IDirect3DRMTexture2::GetDecalOrigin

Retrieves the current origin of the decal.

HRESULT GetDecalOrigin(
LONG * *lplX*,
LONG * *lplY*
);

Parameters

lplX and *lplY*
Addresses of variables that will be filled with the origin of the decal when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::SetDecalOrigin

IDirect3DRMTexture2::GetDecalScale

Retrieves the scaling property of the given decal.

DWORD GetDecalScale( );

Return Values

Returns the scaling property if successful, or -1 otherwise.

See Also

IDirect3DRMTexture2::SetDecalScale

IDirect3DRMTexture2::GetDecalSize

Retrieves the size of the decal.

HRESULT GetDecalSize(
D3DVALUE **lprvWidth*,
D3DVALUE **lprvHeight*
);

Parameters

lprvWidth and *lprvHeight*
Addresses of variables that will be filled with the width and height of the decal when the method returns.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::SetDecalSize

IDirect3DRMTexture2::GetDecalTransparency

Retrieves the transparency property of the decal.

BOOL GetDecalTransparency( );

Return Values

Returns TRUE if the decal has a transparent color, FALSE otherwise.

See Also

IDirect3DRMTexture2::SetDecalTransparency

IDirect3DRMTexture2::GetDecalTransparentColor

Retrieves the transparent color of the decal.

D3DCOLOR GetDecalTransparentColor( );

Return Values

Returns the value of the transparent color.

See Also

IDirect3DRMTexture2::SetDecalTransparentColor

IDirect3DRMTexture2::GetImage

Returns an address of the image that the texture was created with.

D3DRMIMAGE * GetImage( );

Return Values

Returns the address of the D3DRMIMAGE structure that the current Direct3DRMTexture2 object was created with.

IDirect3DRMTexture2::GetShades

Retrieves the number of shades used for each color in the texture when rendering.

DWORD GetShades( );

Return Values

Returns the number of shades.

See Also

IDirect3DRMTexture2::SetShades

IDirect3DRMTexture2::InitFromFile

Initializes a Direct3DRMTexture2 object using the information in a given file.

HRESULT InitFromFile(
LPCSTR *filename*
);

Parameters

filename
A string that specifyies the file from which the texture initialization information is drawn.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You must have created the Direct3DRMTexture2 object to be initialized using the IDirect3DRM2::CreateTexture or IDirect3DRM2::CreateTextureFromSurface methods.

See Also

IDirect3DRMTexture2::InitFromImage, IDirect3DRMTexture2::InitFromResource2, IDirect3DRMTexture2::InitFromSurface

IDirect3DRMTexture2::InitFromImage

Initializes a texture from an image in memory.

HRESULT InitFromImage(
LPD3DRMIMAGE *lpImage*
);

Parameters

lpImage
Address of a D3DRMIMAGE structure describing the source of the texture.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You must have created the IDirect3DRMTexture2 object to be initialized using the IDirect3DRM2::CreateTexture or IDirect3DRM2::CreateTextureFromSurface methods.

See Also

IDirect3DRMTexture2::InitFromFile, IDirect3DRMTexture2::InitFromResource2, IDirect3DRMTexture2::InitFromSurface

IDirect3DRMTexture2::InitFromResource2

Initializes a Direct3DRMTexture2 object from a specified resource.

HRESULT InitFromResource2(
HModule *hModule*,
LPCTSTR *strName*,
LPCTSTR *strType*
);

Parameters

hModule
Handle of the specified resource.

strName
The name of the resource to be used to initialize the texture.

strType
The type name of the resource used to initialize the texture. Textures can be stored in RT_BITMAP and RT_RCDATA resource types, or user-defined types. If the resource type is user-defined, this method passes the resource module handle, the resource name, and the resource type to the **FindResource** Win32 API.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You must have created the IDirect3DRMTexture2 object to be initialized using the IDirect3DRM2::CreateTexture or IDirect3DRM2::CreateTextureFromSurface methods.

See Also

IDirect3DRMTexture2::InitFromFile, IDirect3DRMTexture2::InitFromImage, IDirect3DRMTexture2::InitFromSurface

IDirect3DRMTexture2::InitFromSurface

Initializes a Direct3DRMTexture2 object by using the data from a given DirectDraw surface. This method performs the same as **IDirect3DRMTexture::InitFromSurface**.

HRESULT InitFromSurface(
    LPDIRECTDRAWSURFACE *lpDDS*
    );

Parameters

lpDDS
Address of a DirectDraw surface from which initialization information is drawn.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

You must have created the IDirect3DRMTexture2 object to be initialized using the IDirect3DRM2::CreateTexture or IDirect3DRM2::CreateTextureFromSurface methods.

See Also

IDirect3DRMTexture2::InitFromFile, IDirect3DRMTexture2::InitFromImage, IDirect3DRMTexture2::InitFromResource2

IDirect3DRMTexture2::SetColors

Sets the maximum number of colors used for rendering a Direct3DRMTexture2 object. This method is required only in the ramp color model.

HRESULT SetColors(
    DWORD *ulColors*
    );

Parameters

ulColors
Number of colors. The default value is 8.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::GetColors

IDirect3DRMTexture2::SetDecalOrigin

Sets the origin of the decal as an offset from the top left of the decal.

HRESULT SetDecalOrigin(
LONG *lX*,
LONG *lY*
);

Parameters

lX and *lY*
New origin, in decal coordinates, for the decal. The default origin is [0, 0].

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

This method is also used to add a decal origin key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture2::GetDecalOrigin

IDirect3DRMTexture2::SetDecalScale

Sets the scaling property for a decal.

HRESULT SetDecalScale(
    DWORD *dwScale*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::GetDecalScale

IDirect3DRMTexture2::SetDecalSize

Sets the size of the decal to be used if the decal is being scaled according to its depth in the scene.

HRESULT SetDecalSize(
D3DVALUE *rvWidth*,
D3DVALUE *rvHeight*
);

Parameters

rvWidth and *rvHeight*
New width and height, in model coordinates, of the decal. The default size is [1, 1].

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a decal size key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture2::GetDecalSize

IDirect3DRMTexture2::SetDecalTransparency

Sets the transparency property of the decal.

HRESULT SetDecalTransparency(
    BOOL *bTransp*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::GetDecalTransparency

IDirect3DRMTexture2::SetDecalTransparentColor

Sets the transparent color for a decal.

HRESULT SetDecalTransparentColor(
D3DCOLOR *rcTransp*
);

Parameters

rcTransp
New transparent color. The default transparent color is black.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a decal transparent color key to a Direct3DRMTextureInterpolator object.

See Also

IDirect3DRMTexture2::GetDecalTransparentColor

IDirect3DRMTexture2::SetShades

Sets the maximum number of shades to use for each color for the Direct3DRMTexture2 object when rendering. This method is required only in the ramp color model.

HRESULT SetShades(
    DWORD *ulShades*
    );

Parameters

ulShades
New number of shades. This value must be a power of 2. The default value is 16.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMTexture2::GetShades

IDirect3DRMUserVisual

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 section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMUserVisual Interface.

The **IDirect3DRMUserVisual** interface supports the Init method.

The **IDirect3DRMUserVisual** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef

QueryInterface

Release

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::Init

Initializes a Direct3DRMUserVisual object.

HRESULT Init(
D3DRMUSERVISUALCALLBACK *d3drmUVProc*,
void * *lpArg*
);

Parameters

d3drmUVProc
Application-defined D3DRMUSERVISUALCALLBACK callback function.

lpArg
Application-defined data to be passed to the callback function.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

IDirect3DRMViewport

Applications use the methods of the **IDirect3DRMViewport** interface to work with viewport objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMViewport and IDirect3DRMViewportArray Interface.

The methods of the **IDirect3DRMViewport** interface 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
Miscellaneous	Clear Configure ForceUpdate GetDevice GetDirect3DViewport Pick Render
Offsets	GetX GetY
Projection types	GetProjection SetProjection
Scaling	GetUniformScaling SetUniformScaling
Transformations	InverseTransform Transform

The **IDirect3DRMViewport** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::Clear

Clears the given viewport to the current background color.

HRESULT Clear( );

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::Configure

Reconfigures the origin and dimensions of a viewport.

HRESULT Configure(
LONG *lX*,
LONG *lY*,
DWORD *dwWidth*,
DWORD *dwHeight*
);

Parameters

lX and *lY*
New position of the viewport.

dwWidth and *dwHeight*
New width and height of the viewport.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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

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.

HRESULT ForceUpdate(
DWORD *dwX1*,
DWORD *dwY1*,
DWORD *dwX2*,
DWORD *dwY2*
);

Parameters

dwX1 and *dwY1*
Upper-left corner of area to be updated.

dwX2 and *dwY2*
Lower-right corner of area to be updated.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The system might update any region that is larger than the specified rectangle, including possibly the entire window.

IDirect3DRMViewport::GetBack

Retrieves the position of the back clipping plane for a viewport.

D3DVALUE GetBack( );

Return Values

Returns a value describing the distance between the back clipping plane and the camera.

See Also

IDirect3DRMViewport::SetBack, Viewing Frustum

IDirect3DRMViewport::GetCamera

Retrieves the camera for a viewport.

HRESULT GetCamera(
    LPDIRECT3DRMFRAME **lpCamera*
    );

Parameters

lpCamera
Address of a variable that represents the Direct3DRMFrame object representing the camera.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMViewport::SetCamera, Camera

IDirect3DRMViewport::GetDevice

Retrieves the device associated with a viewport.

HRESULT GetDevice(
    LPDIRECT3DRMDEVICE **lpD3DRMDevice*
    );

Parameters

lpD3DRMDevice
Address of a variable that represents the Direct3DRMDevice object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::GetDirect3DViewport

Retrieves the Direct3D viewport corresponding to the current Direct3DRMViewport.

HRESULT GetDirect3DViewport(
LPDIRECT3DVIEWPORT * *lplpD3DViewport*
);

Parameters

lplpD3DViewport
Address of a pointer that is initialized with a pointer to the Direct3DViewport object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::GetField

Retrieves the field of view for a viewport.

D3DVALUE GetField( );

Return Values

Returns a value describing the field of view.

See Also

IDirect3DRMViewport::SetField, Viewing Frustum

IDirect3DRMViewport::GetFront

Retrieves the position of the front clipping plane for a viewport.

D3DVALUE GetFront( );

Return Values

Returns a value describing the distance from the camera to the front clipping plane.

See Also

IDirect3DRMViewport::SetFront, Viewing Frustum

IDirect3DRMViewport::GetHeight

Retrieves the height, in pixels, of the viewport.

DWORD GetHeight( );

Return Values

Returns the pixel height.

IDirect3DRMViewport::GetPlane

Retrieves the dimensions of the viewport on the front clipping plane.

HRESULT GetPlane(
    D3DVALUE **lpd3dvLeft*,
    D3DVALUE **lpd3dvRight*,
    D3DVALUE **lpd3dvBottom*,
    D3DVALUE **lpd3dvTop*
    );

Parameters

lpd3dvLeft, lpd3dvRight, lpd3dvBottom, and *lpd3dvTop*
Addresses of variables that will be filled to represent the dimensions of the viewport on the front clipping plane.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMViewport::SetPlane

IDirect3DRMViewport::GetProjection

Retrieves the projection type for the viewport. A viewport can use either orthographic or perspective projection.

D3DRMPROJECTIONTYPE GetProjection( );

Return Values

Returns one of the members of the D3DRMPROJECTIONTYPE enumerated type.

See Also

IDirect3DRMViewport::SetProjection

IDirect3DRMViewport::GetUniformScaling

Retrieves the scaling property used to scale the viewing volume into the larger dimension of the window.

BOOL GetUniformScaling( );

Return Values

Returns TRUE if the viewport scales uniformly, or FALSE otherwise.

See Also

IDirect3DRMViewport::SetUniformScaling

IDirect3DRMViewport::GetWidth

Retrieves the width, in pixels, of the viewport.

DWORD GetWidth( );

Return Values

Returns the pixel width.

IDirect3DRMViewport::GetX

Retrieves the x-offset of the start of the viewport on a device.

LONG GetX( );

Return Values

Returns the x-offset.

IDirect3DRMViewport::GetY

Retrieves the y-offset of the start of the viewport on a device.

LONG GetY( );

Return Values

Returns the y-offset.

IDirect3DRMViewport::Init

Initializes a Direct3DRMViewport object.

HRESULT Init(
LPDIRECT3DRMDEVICE *lpD3DRMDevice*,
LPDIRECT3DRMFRAME *lpD3DRMFrameCamera*,
DWORD *xpos*,
DWORD *ypos*,
DWORD *width*,
DWORD *height*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::InverseTransform

Transforms the vector in the *lprvSrc* parameter in screen coordinates to world coordinates, and returns the result in the *lprvDst* parameter.

HRESULT InverseTransform(
D3DVECTOR * *lprvDst*,
D3DRMVECTOR4D * *lprvSrc*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::Pick

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.

HRESULT Pick(
LONG *lX*,
LONG *lY*,
LPDIRECT3DRMPICKEDARRAY* *lplpVisuals*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::Render

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.

HRESULT Render(
LPDIRECT3DRMFRAME *lpD3DRMFrame*
);

Parameters

lpD3DRMFrame

Address of a variable that represents the Direct3DRMFrame object that represents the frame hierarchy to be rendered.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMViewport::SetBack

Sets the position of the back clipping plane for a viewport.

HRESULT SetBack(
    D3DVALUE *rvBack*
    );

Parameters

rvBack
Distance between the camera and the back clipping plane.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a back clipping plane position key to a Direct3DRMViewportInterpolator object.

See Also

IDirect3DRMViewport::GetBack, IDirect3DRMViewport::SetFront, Viewing Frustum

IDirect3DRMViewport::SetCamera

Sets a camera for a viewport.

HRESULT SetCamera(
    LPDIRECT3DRMFRAME *lpCamera*
    );

Parameters

lpCamera
Address of a variable that represents the Direct3DRMFrame object that represents the camera.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

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.

See Also

IDirect3DRMViewport::GetCamera, Camera

IDirect3DRMViewport::SetField

Sets the field of view for a viewport.

HRESULT SetField(
    D3DVALUE *rvField*
    );

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is also used to add a field of view key to a Direct3DRMViewportInterpolator object.

See Also

IDirect3DRMViewport::GetField, Viewing Frustum

IDirect3DRMViewport::SetFront

Sets the position of the front clipping plane for a viewport.

HRESULT SetFront(
    D3DVALUE *rvFront*
    );

Parameters

rvFront
Distance from the camera to the front clipping plane.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The default position is 1.0. If the value passed is less than or equal to zero, this method returns the D3DRMERR_BADVALUE error.

This method is also used to add a front clipping plane position key to a Direct3DRMViewportInterpolator object.

See Also

IDirect3DRMViewport::GetFront, Viewing Frustum

IDirect3DRMViewport::SetPlane

Sets the dimensions of the viewport on the front clipping plane, relative to the camera's z-axis.

HRESULT SetPlane(
D3DVALUE *rvLeft*,
D3DVALUE *rvRight*,
D3DVALUE *rvBottom*,
D3DVALUE *rvTop*
);

Parameters

rvLeft, rvRight, rvBottom, and *rvTop*
Minimum x, maximum x, minimum y, and maximum y coordinates of the four sides of the viewport.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

Unlike the IDirect3DRMViewport::SetField method, which specifies a centered proportional viewport, this method allows you to specify a viewport of arbitrary proportion and position. For example, this method could be used to construct a sheared viewing frustum to implement a right- or left-eye stereo view.

This method is also used to add a plane key to a Direct3DRMViewportInterpolator object.

See Also

IDirect3DRMViewport::GetPlane, IDirect3DRMViewport::SetField

IDirect3DRMViewport::SetProjection

Sets the projection type for a viewport.

HRESULT SetProjection(
    D3DRMPROJECTIONTYPE *rptType*
    );

Parameters

rptType
One of the members of the D3DRMPROJECTIONTYPE enumerated type.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRMViewport::GetProjection

IDirect3DRMViewport::SetUniformScaling

Sets the scaling property used to scale the viewing volume into the larger dimension of the window.

HRESULT SetUniformScaling(
BOOL *bScale*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

This method is typically used with the IDirect3DRMViewport::SetPlane method to support banding.

See Also

IDirect3DRMViewport::GetUniformScaling

IDirect3DRMViewport::Transform

Transforms the vector in the *lprvSrc* parameter in world coordinates to screen coordinates, and returns the result in the *lprvDst* parameter.

HRESULT Transform(
D3DRMVECTOR4D * *lprvDst***,**
D3DVECTOR * *lprvSrc*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

Remarks

The result of the transformation is a four-element homogeneous vector. 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

Applications use the methods of the **IDirect3DRMWinDevice** interface to respond to window messages in a window procedure. This section is a reference to the methods of this interface. For a conceptual overview, see Window Management.

The **IDirect3DRMWinDevice** interface supports the following methods:

HandleActivate
HandlePaint

The **IDirect3DRMWinDevice** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::HandleActivate

Responds to a Windows WM_ACTIVATE message. This ensures that the colors are correct in the active rendering window.

HRESULT HandleActivate(
    WORD *wParam*
    );

Parameters

wParam

WPARAM parameter passed to the message-processing procedure with the WM_ACTIVATE message.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMWinDevice::HandlePaint

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.

HRESULT HandlePaint(
    HDC *hDC*
    );

Parameters

hDC
Handle of the device context (DC).

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

IDirect3DRMWrap

Applications use the methods of the **IDirect3DRMWrap** interface to work with wrap objects. This section is a reference to the methods of this interface. For a conceptual overview, see IDirect3DRMWrap Interface.

The methods of the **IDirect3DRMWrap** interface can be organized into the following groups:

Initialization	Init
Wrap	Apply ApplyRelative

The **IDirect3DRMWrap** interface, like all COM interfaces, inherits the *IUnknown* interface methods. The *IUnknown* interface supports the following three methods:

AddRef
QueryInterface
Release

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::Apply

Applies a Direct3DRMWrap object to its destination object. The destination object is typically a face or a mesh.

HRESULT Apply(
    LPDIRECT3DRMOBJECT *lpObject*
    );

Parameters

lpObject
Address of the destination object.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRM::CreateWrap

IDirect3DRMWrap::ApplyRelative

Applies the wrap to the vertices of the object, first transforming each vertex by the frame's world transformation and the inverse world transformation of the wrap's reference frame.

HRESULT ApplyRelative(
LPDIRECT3DRMFRAME *frame*,
LPDIRECT3DRMOBJECT *mesh*
);

Parameters

frame
Direct3DRMFrame object containing the object to wrap.

mesh
Direct3DRMWrap object to apply.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRM::CreateWrap

IDirect3DRMWrap::Init

Initializes a Direct3DRMWrap object.

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*
);

Parameters

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.

Return Values

Returns D3DRM_OK if successful, or an error otherwise. For a list of possible return codes, see Direct3D Retained-Mode Return Values.

See Also

IDirect3DRM::CreateWrap

D3DRMBOX

Defines the bounding box retrieved by the IDirect3DRMMesh::GetBox and IDirect3DRMMeshBuilder::GetBox methods.

```
typedef struct _D3DRMBOX {
    D3DVECTOR min, max;
}D3DRMBOX;
typedef D3DRMBOX *LPD3DRMBOX;
```

Members

min and **max**
Values defining the bounds of the box. These values are **D3DVECTOR** structures.

See Also

D3DVECTOR, IDirect3DRMMesh::GetBox, IDirect3DRMMeshBuilder::GetBox

D3DRMIMAGE

Describes an image that is attached to a texture by the IDirect3DRM::CreateTexture method. IDirect3DRMTexture::GetImage returns the address of this image.

```
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;
```

Members

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 scan line. 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.

See Also

IDirect3DRM::CreateTexture, IDirect3DRMTexture::GetImage

D3DRMLOADMEMORY

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.

```
typedef struct _D3DRMLOADMEMORY {
    LPVOID lpMemory;
    DWORD  dSize;
} D3DRMLOADMEMORY, *LPD3DRMLOADMEMORY;
```

Members

lpMemory
Address of a block of memory to be loaded.

dSize
Size, in bytes, of the block of memory to be loaded.

See Also

IDirect3DRM::Load, IDirect3DRMAnimationSet::Load, IDirect3DRMFrame::Load, IDirect3DRMMeshBuilder::Load, D3DRMLOADOPTIONS, D3DRMLOADRESOURCE

D3DRMLOADRESOURCE

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.

```
typedef struct _D3DRMLOADRESOURCE {
    HMODULE hModule;
    LPCTSTR lpName;
    LPCTSTR lpType;
} D3DRMLOADRESOURCE, *LPD3DRMLOADRESOURCE;
```

Members

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.

Remarks

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.

See Also

IDirect3DRM::Load, IDirect3DRMAnimationSet::Load, IDirect3DRMFrame::Load, IDirect3DRMMeshBuilder::Load, D3DRMLOADMEMORY, D3DRMLOADOPTIONS

D3DRMPALETTEENTRY

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.)

```
typedef struct _D3DRMPALETTEENTRY {
    unsigned char red;
    unsigned char green;
    unsigned char blue;
    unsigned char flags;
}D3DRMPALETTEENTRY;
typedef D3DRMPALETTEENTRY, *LPD3DRMPALETTEENTRY;
```

Members

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.

See Also

D3DRMIMAGE, D3DRMPALETTEFLAGS

D3DRMPICKDESC

Contains the pick position and face and group identifiers of the objects retrieved by the IDirect3DRMPickedArray::GetPick method.

```
typedef struct _D3DRMPICKDESC {
    ULONG       ulFaceIdx;
    LONG        lGroupIdx;
    D3DVECTOR   vPosition;
} D3DRMPICKDESC, *LPD3DRMPICKDESC;
```

Members

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.

See Also

D3DVECTOR, IDirect3DRMPickedArray::GetPick

D3DRMPICKDESC2

Contains the face and group identifiers, pick position, horizontal and vertical texture coordinates for the vertex, vertex normal, and color of the objects retrieved by the IDirect3DRMPicked2Array::GetPick method.

```
typedef struct _D3DRMPICKDESC2
{
    ULONG ulFaceIdx;
    LONG lGroupIdx;
    D3DVECTOR dvPosition;
    D3DVALUE tu;
    D3DVALUE tv;
    D3DVECTOR dvNormal;
    D3DCOLOR dcColor;

} D3DRMPICKDESC2, *LPD3DRMPICKDESC2;
```

Members

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.

tu and **tv**
Horizontal and vertical texture coordinates, respectively, for the vertex.

dvNormal
Normal vector for the vertex.

dcColor
Vertex color.

D3DRMPMESHLOADSTATUS

Contains the loading status of a progressive mesh loaded with the IDirect3DRMProgressiveMesh::Load method. This structure can be retrieved with the IDirect3DRMProgressiveMesh::GetLoadStatus method.

```
typedef struct _D3DRMPMESHLOADSTATUS {
    DWORD dwSize;
    DWORD dwPMeshSize;
    DWORD dwBaseMeshSize;
    DWORD dwBytesLoaded;
    DWORD dwVerticesLoaded;
    DWORD dwFacesLoaded;
    DWORD dwFlags;
} D3DRMPMESHLOADSTATUS;
typedef D3DRMPMESHLOADSTATUS *LPD3DRMPMESHLOADSTATUS;
```

Members

dwSize
Size of the structure.

dwPMeshSize
Total size of the progressive mesh in bytes.

dwBaseMeshSize
Size of the base mesh in bytes.

dwBytesLoaded
Total number of bytes loaded.

dwVerticesLoaded
Number of vertices loaded.

dwFacesLoaded
Number of faces loaded.

dwFlags
Flags that indicate the status of the progressive mesh loading. Can be one of the following values:

D3DRMPMESHSTATUS_VALID – The progressive mesh object contains valid data.

D3DRMPMESHSTATUS_INTERRUPTED – The download was interrupted either because the application called IDirect3DRMProgressiveMesh::Abort or because the connection was lost.

D3DRMPMESHSTATUS_BASEMESHCOMPLETE – The base mesh has been downloaded.

D3DRMPMESHSTATUS_COMPLETE – All data has been downloaded.

D3DRMPMESHSTATUS_RENDERABLE – It is now possible to render the mesh.

See Also

IDirect3DRMProgressiveMesh::GetLoadStatus, IDirect3DRMProgressiveMesh::Load

D3DRMQUATERNION

Describes the rotation used by the IDirect3DRMAnimation::AddRotateKey method, and the quaternion used in IDirect3DRMFrame2::SetQuaternion. It is also used in several of Direct3D's mathematical functions.

```
typedef struct _D3DRMQUATERNION {
    D3DVALUE   s;
    D3DVECTOR  v;
}D3DRMQUATERNION;
typedef D3DRMQUATERNION, *LPD3DRMQUATERNION;
```

See Also

IDirect3DRMAnimation::AddRotateKey, IDirect3DRMFrame2::SetQuaternion, D3DRMQuaternionFromRotation, D3DRMQuaternionMultiply, D3DRMQuaternionSlerp, D3DRMMatrixFromQuaternion

D3DRMRAY

Defines the direction and starting position of the ray in IDirect3DRMFrame2::RayPick.

```
typedef struct _D3DRMRAY
{    D3DVECTOR dvDir;
     D3DVECTOR dvPos;
} D3DRMRAY, *LPD3DRMRAY;
```

Members

dvDir

The direction of the ray used in a ray pick.

dvPos

The starting position of the ray used in a ray pick.

D3DRMVECTOR4D

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.

```
typedef struct _D3DRMVECTOR4D {
    D3DVALUE x;
    D3DVALUE y;
    D3DVALUE z;
    D3DVALUE w;
}D3DRMVECTOR4D;
typedef D3DRMVECTOR4D, *LPD3DRMVECTOR4D;
```

Members

x, **y**, **z**, and **w**
Values of the **D3DVALUE** type describing homogeneous values. These values define the result of the transformation.

See Also

IDirect3DRMViewport::Transform, IDirect3DRMViewport::InverseTransform

D3DRMVERTEX

Describes a vertex in a Direct3DRMMesh object.

```
typedef struct _D3DRMVERTEX{
    D3DVECTOR position;
    D3DVECTOR normal;
    D3DVALUE  tu, tv;
    D3DCOLOR  color;
} D3DRMVERTEX;
```

Members

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.

See Also

IDirect3DRMMesh::GetVertices, IDirect3DRMMesh::SetVertices

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.

```
typedef enum _D3DRMCOLORSOURCE{
    D3DRMCOLOR_FROMFACE,
    D3DRMCOLOR_FROMVERTEX
} D3DRMCOLORSOURCE;
```

Values

D3DRMCOLOR_FROMFACE
The object's color source is a face.

D3DRMCOLOR_FROMVERTEX
The object's color source is a vertex.

See Also

IDirect3DRMMeshBuilder::SetColorSource, IDirect3DRMMeshBuilder::GetColorSource

D3DRMCOMBINETYPE

Specifies how to combine two matrices.

```
typedef enum _D3DRMCOMBINETYPE{
    D3DRMCOMBINE_REPLACE,
    D3DRMCOMBINE_BEFORE,
    D3DRMCOMBINE_AFTER
} D3DRMCOMBINETYPE;
```

Values

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.

Remarks

The order of the supplied and current matrices when they are multiplied together is important because matrix multiplication is not commutative.

See Also

IDirect3DRMFrame::AddRotation, IDirect3DRMFrame::AddScale, IDirect3DRMFrame::AddTransform, IDirect3DRMFrame::AddTranslation

D3DRMFILLMODE

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

```
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;
```

Values

D3DRMFILL_POINTS
Fills points only; minimum fill mode.

D3DRMFILL_WIREFRAME
Fill wire frames.

D3DRMFILL_SOLID
Fill solid objects.

D3DRMFILL_MASK
Fill using a mask.

D3DRMFILL_MAX
Maximum value for fill mode.

See Also

D3DRMLIGHTMODE, D3DRMSHADEMODE, D3DRMRENDERQUALITY

D3DRMFOGMODE

Contains values that specify how rapidly and in what ways the fog effect intensifies with increasing distance from the camera.

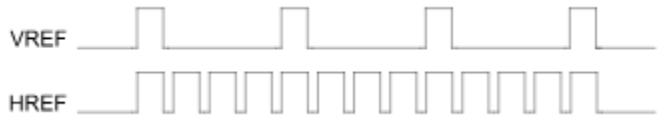
In monochromatic (ramp) lighting mode, fog works properly only when the fog color is black. (If there is no lighting, any fog color will work, since in this case any fog color is effectively black.)

```
typedef enum _D3DRMFOGMODE{
    D3DRMFOG_LINEAR,
    D3DRMFOG_EXPONENTIAL,
    D3DRMFOG_EXPONENTIALSQUARED
} D3DRMFOGMODE;
```

Values

D3DRMFOG_LINEAR

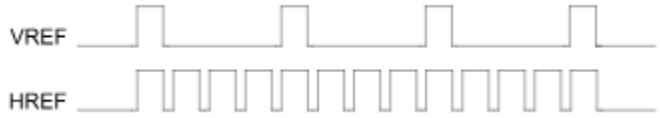
The fog effect intensifies linearly between the start and end points, according to the following formula:



This is the only fog mode currently supported.

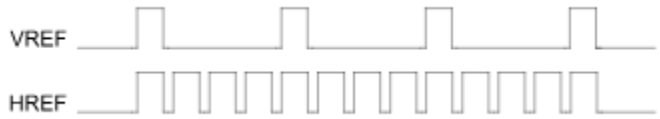
D3DRMFOG_EXPONENTIAL

The fog effect intensifies exponentially, according to the following formula:



D3DRMFOG_EXPONENTIALSQUARED

The fog effect intensifies exponentially with the square of the distance, according to the following formula:



Remarks

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.

See Also

IDirect3DRMFrame::SetSceneFogMode, IDirect3DRMFrame::SetSceneFogParams

D3DRMFRAMECONSTRAINT

Describes the axes of rotation to constrain when viewing a Direct3DRMFrame object. The IDirect3DRMFrame::LookAt method uses this enumerated type.

```
typedef enum _D3DRMFRAMECONSTRAINT {
    D3DRMCONSTRAIN_Z,
    D3DRMCONSTRAIN_Y,
    D3DRMCONSTRAIN_X
} D3DRMFRAMECONSTRAINT;
```

Values

D3DRMCONSTRAIN_Z
Use only x and y rotations.

D3DRMCONSTRAIN_Y
Use only x and z rotations.

D3DRMCONSTRAIN_X
Use only y and z rotations.

See Also

IDirect3DRMFrame::LookAt

D3DRMLIGHTMODE

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

```
typedef enum _D3DRMLIGHTMODE {
    D3DRMLIGHT_OFF      = 0 * D3DRMSHADE_MAX,
    D3DRMLIGHT_ON       = 1 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MASK     = 7 * D3DRMSHADE_MAX,
    D3DRMLIGHT_MAX      = 8 * D3DRMSHADE_MAX
} D3DRMLIGHTMODE;
```

Values

D3DRMLIGHT_OFF
Lighting is off.

D3DRMLIGHT_ON
Lighting is on.

D3DRMLIGHT_MASK
Lighting uses a mask.

D3DRMLIGHT_MAX
Maximum lighting mode.

See Also

D3DRMFILLMODE, D3DRMSHADEMODE, D3DRMRENDERQUALITY

D3DRMLIGHTTYPE

Defines the light type in calls to the IDirect3DRM::CreateLight method.

```
typedef enum _D3DRMLIGHTTYPE{
    D3DRMLIGHT_AMBIENT,
    D3DRMLIGHT_POINT,
    D3DRMLIGHT_SPOT,
    D3DRMLIGHT_DIRECTIONAL,
    D3DRMLIGHT_PARALLELPOINT
} D3DRMLIGHTTYPE;
```

Values

D3DRMLIGHT_AMBIENT
Light is an ambient source.

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 point source.

D3DRMMATERIALMODE

Describes the source of material information for visuals rendered with a frame.

```
typedef enum _D3DRMMATERIALMODE{
    D3DRMMATERIAL_FROMMESH,
    D3DRMMATERIAL_FROMPARENT,
    D3DRMMATERIAL_FROMFRAME
} D3DRMMATERIALMODE;
```

Values

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.

See Also

IDirect3DRMFrame::GetMaterialMode, IDirect3DRMFrame::SetMaterialMode

D3DRMPALETTEFLAGS

Used to define how a color may be used in the D3DRMPALETTEENTRY structure.

```
typedef enum _D3DRMPALETTEFLAGS {
    D3DRMPALETTE_FREE,
    D3DRMPALETTE_READONLY,
    D3DRMPALETTE_RESERVED
} D3DRMPALETTEFLAGS;
```

Values

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.

See Also

D3DRMPALETTEENTRY

D3DRMPROJECTIONTYPE

Defines the type of projection used in a Direct3DRMViewport object. The IDirect3DRMViewport::GetProjection and IDirect3DRMViewport::SetProjection methods use this enumerated type. The right-hand types enable right-handed projection.

The axes of the camera (see IDirect3DRMFrame2::SetAxes) are used in both left-handed and right-handed projection to determine what direction the camera is facing.

```
typedef enum _D3DRMPROJECTIONTYPE{
    D3DRMPROJECT_PERSPECTIVE,
    D3DRMPROJECT_ORTHOGRAPHIC,
    D3DRMPROJECT_RIGHTHANDPERSPECTIVE,
    D3DRMPROJECT_RIGHTHANDORTHOGRAPHIC
} D3DRMPROJECTIONTYPE;
```

Values

D3DRMPROJECT_PERSPECTIVE
The projection is perspective and left-handed.

D3DRMPROJECT_ORTHOGRAPHIC
The projection is orthographic and left-handed.

D3DRMPROJECT_RIGHTHANDPERSPECTIVE
The projection is perspective and right-handed.

D3DRMPROJECT_RIGHTHANDORTHOGRAPHIC
The projection is orthographic and right-handed.

See Also

IDirect3DRMViewport::GetProjection, IDirect3DRMViewport::SetProjection

D3DRMRENDERQUALITY

Combines descriptions of the shading mode, lighting mode, and filling mode for a Direct3DRMMesh object.

```
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)
```

Values

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 currently supported.

See Also

IDirect3DRMMesh::GetGroupQuality, IDirect3DRMMesh::SetGroupQuality,IDirect3DRMDevice2::SetRenderMode

D3DRMSHADEMODE

One of the enumerated types that is used in the definition of the D3DRMRENDERQUALITY type.

```
typedef enum _D3DRMSHADEMODE {
    D3DRMSHADE_FLAT     = 0,
    D3DRMSHADE_GOURAUD  = 1,
    D3DRMSHADE_PHONG    = 2,
    D3DRMSHADE_MASK     = 7,
    D3DRMSHADE_MAX      = 8
} D3DRMSHADEMODE;
```

See Also

D3DRMFILLMODE, D3DRMLIGHTMODE, D3DRMRENDERQUALITY

D3DRMSORTMODE

Describes how child frames are sorted in a scene.

```
typedef enum _D3DRMSORTMODE {
    D3DRMSORT_FROMPARENT,
    D3DRMSORT_NONE,
    D3DRMSORT_FRONTTOBACK,
    D3DRMSORT_BACKTOFRONT
} D3DRMSORTMODE;
```

Values

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.

See Also

IDirect3DRMFrame::GetSortMode, IDirect3DRMFrame::SetSortMode

D3DRMTEXTUREQUALITY

Describes how a device interpolates between pixels in a texture and pixels in a viewport. This enumerated type is used by the IDirect3DRMDevice::SetTextureQuality and IDirect3DRMDevice::GetTextureQuality methods.

```
typedef enum _D3DRMTEXTUREQUALITY{
    D3DRMTEXTURE_NEAREST,
    D3DRMTEXTURE_LINEAR,
    D3DRMTEXTURE_MIPNEAREST,
    D3DRMTEXTURE_MIPLINEAR,
    D3DRMTEXTURE_LINEARMIPNEAREST,
    D3DRMTEXTURE_LINEARMIPLINEAR
} D3DRMTEXTUREQUALITY;
```

Values

D3DRMTEXTURE_NEAREST
Choose the nearest pixel in the texture. Does not support MIP mapping.

D3DRMTEXTURE_LINEAR
Linearly interpolate the four nearest pixels. Does not support MIP mapping.

D3DRMTEXTURE_MIPNEAREST
Similar to D3DRMTEXTURE_NEAREST, but uses the appropriate mipmap instead of the texture. Pixel sampling and MIP mapping are both nearest.

D3DRMTEXTURE_MIPLINEAR
Similar to D3DRMTEXTURE_LINEAR, but uses the appropriate mipmap instead of the texture. Pixel sampling is linear; MIP mapping is nearest.

D3DRMTEXTURE_LINEARMIPNEAREST
Similar to D3DRMTEXTURE_MIPNEAREST, but interpolates between the two nearest mipmaps. Pixel sampling is nearest; MIP mapping is linear.

D3DRMTEXTURE_LINEARMIPLINEAR
Similar to D3DRMTEXTURE_MIPLINEAR, but interpolates between the two nearest mipmaps. Both pixel sampling and MIP mapping are linear.

D3DRMUSERVISUALREASON

Defines the reason the system has called the D3DRMUSERVISUALCALLBACK callback function.

```
typedef enum _D3DRMUSERVISUALREASON {
    D3DRMUSERVISUAL_CANSEE,
    D3DRMUSERVISUAL_RENDER
} D3DRMUSERVISUALREASON;
```

Values

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.

See Also

D3DRMUSERVISUALCALLBACK

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.

```
typedef enum _D3DRMWRAPTYPE{
    D3DRMWRAP_FLAT,
    D3DRMWRAP_CYLINDER,
    D3DRMWRAP_SPHERE,
    D3DRMWRAP_CHROME
} D3DRMWRAPTYPE;
```

Values

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.

See Also

IDirect3DRM::CreateWrap, IDirect3DRMWrap::Init, IDirect3DRMWrap Interface

D3DRMXOFFORMAT

Defines the file type used by the IDirect3DRMMeshBuilder::Save method.

```
typedef enum _D3DRMXOFFORMAT{
    D3DRMXOF_BINARY,
    D3DRMXOF_COMPRESSED,
    D3DRMXOF_TEXT
} D3DRMXOFFORMAT;
```

Values

D3DRMXOF_BINARY
File is in binary format. This is the default setting.

D3DRMXOF_COMPRESSED
Not currently supported.

D3DRMXOF_TEXT
File is in text format.

Remarks

The D3DRMXOF_BINARY and D3DRMXOF_TEXT settings are mutually exclusive.

See Also

IDirect3DRMMeshBuilder::Save

D3DRMZBUFFERMODE

Describes whether z-buffering is enabled.

```
typedef enum _D3DRMZBUFFERMODE {
    D3DRMZBUFFER_FROMPARENT,
    D3DRMZBUFFER_ENABLE,
    D3DRMZBUFFER_DISABLE
} D3DRMZBUFFERMODE;
```

Values

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.

See Also

IDirect3DRMFrame::GetZbufferMode, IDirect3DRMFrame::SetZbufferMode

D3DRMANIMATIONOPTIONS

Specifies values used by the IDirect3DRMAnimation::GetOptions and IDirect3DRMAnimation::SetOptions methods to define how animations are played.

```
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
```

Parameters

D3DRMANIMATION_CLOSED

The animation plays continually, looping back to the beginning whenever it reaches the end. In a closed animation, the last key in the animation should be a repeat of the first. This repeated key is used to indicate the time difference between the last and first keys in the looping animation.

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

Describes the color model implemented by the device. For more information, see the **D3DCOLORMODEL** enumerated type.

```
typedef D3DCOLORMODEL D3DRMCOLORMODEL;
```

See Also

D3DCOLORMODEL

D3DRMINTERPOLATIONOPTIONS

Defines options for the IDirect3DRMInterpolator::Interpolate method. These options modify how the object is loaded.

```
typedef DWORD D3DRMINTERPOLATIONOPTIONS;
#define D3DRMINTERPOLATION_OPEN 0x01L
#define D3DRMINTERPOLATION_CLOSED 0x02L
#define D3DRMINTERPOLATION_NEAREST 0x0100L
#define D3DRMINTERPOLATION_LINEAR 0x04L
#define D3DRMINTERPOLATION_SPLINE 0x08L
#define D3DRMINTERPOLATION_VERTEXCOLOR 0x40L
#define D3DRMINTERPOLATION_SLERPNORMALS 0x80L
```

Parameters

D3DRMINTERPOLATION_OPEN
The first and last keys of each key chain will fix the interpolated values outside of the index span.

D3DRMINTERPOLATION_CLOSED
The interpolation is cyclic. The keys effectively repeat infinitely with a period equal to the index span. For compatibility with animations, any key with an index equal to the end of the span is ignored.

D3DRMINTERPOLATION_NEAREST
Nearest key value is used for in-betweening on each key chain.

D3DRMINTERPOLATION_LINEAR
Linear interpolation between the two nearest keys is used for in-betweening on each key chain.

D3DRMINTERPOLATION_SPLINE
A B-spline blending function on the 4 nearest keys is used for in-betweening on each key chain.

D3DRMINTERPOLATION_VERTEXCOLOR
Specifies that vertex colors should be interpolated. Only affects the interpolation of IDirect3DRMMesh::SetVertices.

D3DRMINTERPOLATION_SLERPNORMALS
Specifies that vertex normals should be spherically interpolated (not currently implemented). Only affects the interpolation of IDirect3DRMMesh::SetVertices.

D3DRMLOADOPTIONS

Defines options for the IDirect3DRM::Load, IDirect3DRMAnimationSet::Load, IDirect3DRMFrame::Load, IDirect3DRMMeshBuilder::Load methods, and IDirect3DRMProgressiveMesh::Load methods. These options modify how the object is loaded.

```
typedef DWORD D3DRMLOADOPTIONS;
#define D3DRMLOAD_FROMFILE              0x00L
#define D3DRMLOAD_FROMRESOURCE          0x01L
#define D3DRMLOAD_FROMMEMORY            0x02L
#define D3DRMLOAD_FROMURL               0x08L
#define D3DRMLOAD_BYNAME                0x10L
#define D3DRMLOAD_BYPOSITION            0x20L
#define D3DRMLOAD_BYGUID                0x30L
#define D3DRMLOAD_FIRST                 0x40L
#define D3DRMLOAD_INSTANCEBYREFERENCE   0x100L
#define D3DRMLOAD_INSTANCEBYCOPYING     0x200L
#define D3DRMLOAD_ASYNCHRONOUS          0x400L
```

Parameters

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_FROMURL

Load from a URL.

Identifier flags

D3DRMLOAD_BYNAME

Load any object by using a specified name.

D3DRMLOAD_BYPOSITION

Load a stand-alone object based on a given zero-based position (that is, the *n*th object in the file). Stand-alone objects can contain other objects, but are not contained by any other objects.

D3DRMLOAD_BYGUID

Load any object by using a specified globally unique identifier (GUID).

D3DRMLOAD_FIRST

This is the default setting. Load the first stand-alone object of the given type (for example, a mesh if the application calls IDirect3DRMMeshBuilder::Load). Stand-alone objects can contain other objects, but are not contained by any other objects.

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.

Source flags

D3DRMLOAD_ASYNCHRONOUS

The **Load** call will return immediately. It is up to the application to use events to find out how the load is progressing. By default, loading is done synchronously and the **Load** call will not return until all data has been loaded or an error occurs.

Remarks

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

Specifies values used by the IDirect3DRMMesh::GetGroupMapping and IDirect3DRMMesh::SetGroupMapping methods to define how textures are mapped to a group.

```
typedef DWORD D3DRMMAPPING, D3DRMMAPPINGFLAG;
static const D3DRMMAPPINGFLAG D3DRMMAP_WRAPU = 1;
static const D3DRMMAPPINGFLAG D3DRMMAP_WRAPV = 2;
static const D3DRMMAPPINGFLAG D3DRMMAP_PERSPCORRECT = 4;
```

Parameters

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.

Remarks

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 IDirect3DRMWrap Interface.

See Also

IDirect3DRMWrap Interface, IDirect3DRMMesh::GetGroupMapping, IDirect3DRMMesh::SetGroupMapping

D3DRMMATRIX4D

Expresses a transformation as an array. The organization of the matrix entries is D3DRMMATRIX4D[*row*][*column*].

```
typedef D3DVALUE D3DRMMATRIX4D[4][4];
```

See Also

IDirect3DRMFrame::AddTransform, IDirect3DRMFrame::GetTransform

D3DRMSAVEOPTIONS

Defines options for the IDirect3DRMMeshBuilder::Save method.

```
typedef DWORD D3DRMSAVEOPTIONS;
#define D3DRMXOFSAVE_NORMALS 1
#define D3DRMXOFSAVE_TEXTURECOORDINATES 2
#define D3DRMXOFSAVE_MATERIALS 4
#define D3DRMXOFSAVE_TEXTURENAMES 8
#define D3DRMXOFSAVE_ALL 15
#define D3DRMXOFSAVE_TEMPLATES 16
#define D3DRMSAVE_TEXTURETOPOLOGY 32
```

Parameters

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.

D3DRMXOFSAVE_TEMPLATES
Save templates with the file. By default, templates are not saved.

D3DRMSAVE_TEXTURETOPOLOGY
Save the mesh's face wrap values (set by IDirect3DRMMeshBuilder::SetTextureTopology or IDirect3DRMFace::SetTextureTopology). This flag is not included when you pass D3DRMXOFSAVE_ALL to the **Save** method. You should pass D3DRMSAVE_TEXTURETOPOLOGY|D3DRMXOFSAVE_ALL if you really want to save everything. You only need to pass this flag when you've called IDirect3DRMMeshBuilder::SetTextureTopology or IDirect3DRMFace::SetTextureTopology) calls and want to preserve the values you set.

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_BADPMDATA

The data in the X File is corrupted. The conversion to a progressive mesh succeeded but produced an invalid progressive mesh in the X File.

D3DRMERR_BADTYPE

Bad argument type passed.

D3DRMERR_BADVALUE

Bad argument value passed.

D3DRMERR_BOXNOTSET

An attempt was made to access a bounding box (for example, with IDirect3DRMFrame2::GetBox) when no bounding box was set on the frame.

D3DRMERR_CONNECTIONLOST

Data connection was lost during a load, clone, or duplicate.

D3DRMERR_FACEUSED

Face already used in a mesh.

D3DRMERR_FILENOTFOUND

File cannot be opened.

D3DRMERR_INVALIDDATA

The method received or accessed data that is invalid.

D3DRMERR_INVALIDOBJECT

The method received a pointer to an object that is invalid.

D3DRMERR_INVALIDPARAMS

One of the parameters passed to the method is invalid.

D3DRMERR_NOTDONEYET

Unimplemented.

D3DRMERR_NOTENOUGHDATA

Not enough data has been loaded to perform the requested operation.

D3DRMERR_NOTFOUND

Object not found in specified place.

D3DRMERR_PENDING

The data required to supply the requested information has not finished loading.

D3DRMERR_REQUESTTOOLARGE

An attempt was made to set a level of detail in a progressive mesh greater than the maximum available.

D3DRMERR_REQUESTTOOSMALL

An attempt was made to set the minimum rendering detail of a progressive mesh smaller than the detail in the base mesh (the minimum for rendering).

D3DRMERR_UNABLETOEXECUTE

Unable to carry out procedure.

Copyright Notification

Introduction to File Formats

This document specifies the file format introduced with DirectX 2. A binary version of this format was subsequently released with DirectX 3 and this is also detailed in this reference.

The DirectX File Format provides a rich template-driven file format that enables the storage of meshes, textures, animations and user-definable objects. Support for animation sets allows predefined paths to be stored for playback in real time. Also supported are instancing which allows multiple references to an object, such as a mesh, while storing its data only once per file, and hierarchies to express relationships between data records. The DirectX file format is used natively by the Direct3D Retained Mode API, providing support for reading predefined objects into an application or writing mesh information constructed by the application in real time.

The File Format provides low-level data primitives, upon which applications define higher level primitives via templates. This is the method by which Direct3D defines higher level primitives such as Vectors, Meshes, Matrices and Colors.

This document details the API independent features of the file format and also the method by which Direct3D Retained Mode uses the file format.

File Format Architecture

The DirectX file format is an architecture- and context-free file format. It is template driven and is free of any usage knowledge. The file format may be used by any client application and currently is used by Direct3D Retained Mode to describe geometry data, frame hierarchies and animations.

The rest of this section will deal with the content and syntax of the file format. The file format uses the extension .X when used with the DirectX SDK.

Reserved Words

The following words are reserved and must not be used:

- ARRAY
- BYTE
- CHAR
- CSTRING
- DOUBLE
- DWORD
- FLOAT
- STRING
- TEMPLATE
- UCHAR
- UNICODE
- WORD

Header

The variable length header is compulsory and must be at the beginning of the data stream. The header contains the following

Type	Sub Type	Size	Contents	Content Meaning
Magic Number - required		4 bytes	"xof "	
Version Number - required	Major Number	2 bytes	03	Major version 3
	Minor Number	2 bytes	02	Minor version 2
Format Type - required		4 bytes	"txt "	Text File
			"bin "	Binary File
			"com "	Compressed File
Compression Type - required		4 bytes	"lzw "	
if format type is compressed			"zip "	
			etc...	
Float size - required		4 bytes	0064	64 bit floats
			0032	32 bit floats

Example

```
xof 0302txt 0064
```

Comments

Comments are only applicable in text files. Comments may occur anywhere in the data stream. A comment begins with either C++ style double-slashes "//", or a hash character "#". The comment runs to the next new-line.

```
# This is a comment.
// This is another comment.
```

Templates

Templates define how the data stream is interpreted - the data is modulated by the template definition. A template has the following form:

```
template <template-name> {
<UUID>
<member 1>;
...
<member n>;
[restrictions]
}
```

This section discusses the following parts of a template:

- Template name
- UUID
- Members
- Restrictions

Example templates are presented in Examples.

Template name

This is an alphanumeric name which may include the underscore character '_'. It must not begin with a digit.

UUID

A universally unique identifier formatted to the OSF DCE standard and surrounded by angle brackets '<' and '>'. For example: <3D82AB43-62DA-11cf-AB39-0020AF71E433>

Members

Template members consist of a named data type followed by an optional name or an array of a named data type. Valid primitive data types are

Type	Size
WORD	16 bits
DWORD	32 bits
FLOAT	IEEE float
DOUBLE	64 bits
CHAR	8 bits
UCHAR	8 bits
BYTE	8 bits
STRING	NULL terminated string
CSTRING	Formatted C-string (currently unsupported)
UNICODE	UNICODE string (currently unsupported)

Additional data types defined by templates encountered earlier in the data stream may also be referenced within a template definition. No forward references are allowed.

Any valid data type can be expressed as an array in the template definition. The basic syntax is as follows

```
array <data-type> <name>[<dimension-size>];
```

Where <dimension-size> can either be an integer or a named reference to another template member whose value is then substituted.

Arrays may be n-dimensional where n is determined by the number of paired square brackets trailing the statement. For example

```
array DWORD FixedHerd[24];
array DWORD Herd[nCows];
array FLOAT Matrix4x4[4][4];
```

Restrictions

Templates may be *open*, *closed* or *restricted*. These restrictions determine which data types may appear in the immediate hierarchy of a data object defined by the template. An open template has no restrictions, a closed template rejects all data types and a restricted template allows a named list of data types. The syntax is as follows

Three periods enclosed by square brackets indicate an open template

```
[ ((( ]
```

A comma separated list of named data types followed optionally by their uuids enclosed by square brackets indicates a restricted template

```
[ { data-type [ UUID ] , }... ]
```

The absence of either of the above indicates a closed template.

Examples

```
template Mesh {
<3D82AB44-62DA-11cf-AB39-0020AF71E433>
DWORD nVertices;
array Vector vertices[nVertices];
DWORD nFaces;
array MeshFace faces[nFaces];
 [ ... ]                // An open template
}
template Vector {
<3D82AB5E-62DA-11cf-AB39-0020AF71E433>
FLOAT x;
FLOAT y;
FLOAT z;
}                        // A closed template
template FileSystem {
<UUID>
STRING name;
[ Directory <UUID>, File <UUID> ]    // A restricted template
}
```

There is one special template - the *Header* template. It is recommended that each application define such a template and use it to define application specific information such as version information. If present, this header will be read by the File Format API and if a *flags* member is available, it will be used to determine how the following data is interpreted. The *flags* member, if defined, should be a DWORD. One bit is currently defined - bit 0. If this is clear, the following data in the file is binary, if set, the following data is text. Multiple header data objects can be used to switch between binary and text during the file.

Data

Data objects contain the actual data or a reference to that data. Each has a corresponding template that specifies the data type.

Data objects have the following form

```
<Identifier> [name] {
<member 1>;
...
<member n>;
}
```

This section discusses the following parts of data objects:

- Identifier
- Name
- Members

Example templates are presented in Examples.

Identifier

This is compulsory and must match a previously defined data type or primitive.

Name

This is optional. (See above for syntax definition.)

Members

Data members can be one of the following

Data object

A nested data object. This allows the hierarchical nature of the file format to be expressed. The types of nested data objects allowed in the hierarchy may be restricted. See *Templates* above.

Data reference

A reference to a previously encountered data object. The syntax is as follows

```
{ name }
```

Integer list

A semicolon separated list of integers. For example

```
1; 2; 3;
```

Float list

A semicolon separated list of floats. For example

```
1.0; 2.0; 3.0;
```

String list

A semicolon separated list of strings. For example

```
"Moose"; "Goats"; "Sheep";
```

Use of commas and semicolons

This is perhaps the most complex syntax issue in the file format, but is very strict: Commas are used to separate array members; semicolons terminate every data item.

For example, if we have a template defined as

```
template foo {
DWORD bar;
}
```

then an instance of this would look like

```
foo dataFoo {
1;
}
```

Next, we have a template containing another template

```
template foo {
DWORD bar;
DWORD bar2;
}
template container {
FLOAT aFloat;
foo aFoo;
}
```

then an instance of this would look like

```
container dataContainer {
1.1;
2; 3;;
}
```

Note that the second line that represents the **foo** inside container has two semi-colons at the end of the line. The first indicates the end of the data item **aFoo** (inside container), and the second indicates the end of the **container**.

Next we consider arrays.

```
Template foo {
array DWORD bar[3];
}
```

then an instance of this would look like

```
foo aFoo {
1, 2, 3;
}
```

In the array case, there is no need for the data items to be separated by a semi-colon as they are delineated by a comma. The semi-colon at the end marks the end of the array.

Now consider a template that contains an array of data items defined by a template

```
template foo {
DWORD bar;
DWORD bar2;
}
template container {
DWORD count;
array foo fooArray[count];
}
```

then an instance of this would look like

```
container aContainer {
3;
1;2;,3;4;,5;6;;
}
```

Appendix A: Templates

This appendix lists the templates used by Direct3D's Retained Mode. A familiarity with Direct3D Retained mode data types is assumed.

Template Name: Header

UUID

<3D82AB43-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
major	WORD		None
minor	WORD		
flags	DWORD		

Description

This template defines the application specific header for the Direct3D Retained mode usage of the DirectX File Format. The retained mode uses the major and minor flags to specify the current major and minor versions for the retained mode file format.

Template Name: Vector

UUID

<3D82AB5E-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
x	FLOAT		None
y	FLOAT		
z	FLOAT		

Description

This template defines a vector.

Template Name: Coords2d

UUID

<F6F23F44-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
u	FLOAT		None
v	FLOAT		

Description

A two dimensional vector used to define a mesh's texture coordinates.

Template Name: Quaternion

UUID

<10DD46A3-775B-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
s	FLOAT		None
v	Vector		

Description

Currently unused.

Template Name: Matrix4x4

UUID

<F6F23F45-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
matrix	array FLOAT	16	None

Description

This template defines a 4 by 4 matrix. This is used as a frame transformation matrix.

Template Name: ColorRGBA

UUID

<35FF44E0-6C7C-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
red	FLOAT		None
green	FLOAT		
blue	FLOAT		
alpha	FLOAT		

Description

This template defines a color object with an alpha component. This is used for the face color in the material template definition.

Template Name: ColorRGB

UUID

<D3E16E81-7835-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
red	FLOAT		None
green	FLOAT		
blue	FLOAT		

Description

This template defines the basic RGB color object.

Template Name: Indexed Color

UUID

<1630B820-7842-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size
index	DWORD	
ColorRGBA	indexColor	

Description

This template consists of an index parameter and a RGBA color and is used in for defining mesh vertex colors. The index defines the vertex to which the color is applied.

Template Name: Boolean

UUID

<4885AE61-78E8-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
WORD	truefalse		None

Description

Defines a simple boolean type. Should be set to 0 or 1.

Template Name: Boolean2d

UUID

<4885AE63-78E8-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
u	Boolean		None
v	Boolean		

Description

This defines a set of 2 boolean values used in the MeshFaceWraps template in order to define the texture topology of an individual face.

Template Name: Material

UUID

<3D82AB4D-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
faceColor	ColorRGBA		Any
power	FLOAT		
specularColor	ColorRGB		
emissiveColor	ColorRGB		

Description

This template defines a basic material color which can be applied to either a complete mesh or a mesh's individual faces. The power is the specular exponent of the material. Note that the ambient color requires an alpha component.

TextureFilename is an optional data object used by Direct3DRM. If this is not present, the face is untextured.

Template Name: TextureFilename

UUID

<A42790E1-7810-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
filename	STRING		None

Description

This template allows you to specify the filename of a texture to apply to a mesh or a face. This should appear within a material object.

Template Name: MeshFace

UUID

<3D82AB5F-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
nFaceVertexIndices	DWORD		None
faceVertexIndices	array DWORD	nFaceVertexIndicies	

Description

This template is used by the Mesh template to define a mesh's faces. Each element of the nFaceVertexIndices array references a mesh vertex used to build the face.

Template Name: MeshFaceWraps

UUID

<4885AE62-78E8-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nFaceWrapValues	DWORD		None
faceWrapValues	Boolean2d		

Description

This template is used to define the texture topology of each face in a wrap. nFaceWrapValues should be equal to the number of faces in a mesh.

Template Name: MeshTextureCoords

UUID

<F6F23F40-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nTextureCoords	DWORD		None
textureCoords	array Coords2d	nTextureCoords	

Description

This template defines a mesh's texture coordinates.

Template Name: MeshNormals

UUID

<F6F23F43-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nNormals	DWORD		None
normals	array Vector	nNormals	
nFaceNormals	DWORD		
faceNormals	array MeshFace	nFaceNormals	

Description

This template defines normals for a mesh. The first array of vectors are the normal vectors themselves, and the second array is an array of indexes specifying which normals should be applied to a given face. nFaceNormals should be equal to the number of faces in a mesh.

Template Name: MeshVertexColors

UUID

<1630B821-7842-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nVertexColors	DWORD		None
vertexColors	array IndexedColor	nVertexColors	

Description

This template specifies vertex colors for a mesh, as opposed to applying a material per face or per mesh.

Template Name: MeshMaterialList

UUID

<F6F23F42-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nMaterials	DWORD		Material
nFaceIndexes	DWORD		
FaceIndexes	array DWORD	nFaceIndexes	

Description

This template is used in a mesh object to specify which material applies to which faces. nMaterials specifies how many materials are present, and materials specify which material to apply.

Template Name: Mesh

UUID

<3D82AB44-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
nVertices	DWORD		Any
vertices	array Vector	nVertices	
nFaces	DWORD		
faces	array MeshFace	nFaces	

Description

This template defines a simple mesh. The first array is a list of vertices and the second array defines the faces of the mesh by indexing into the vertex array.

Optional Data Elements

The following optional data elements are used by Direct3DRM.

MeshFaceWraps	If this is not present, wrapping for both u and v defaults to false.
MeshTextureCoords	If this is not present, there are no texture coordinates.
MeshNormals	If this is not present, normals are generated using the GenerateNormals() member of the API.
MeshVertexColors	If this is not present, the colors default to white.
MeshMaterialList	If this is not present, the material defaults to white.

Template Name: FrameTransformMatrix

UUID

<F6F23F41-7686-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
frameMatrix	Matrix4x4		None

Description

This template defines a local transform for a frame (and all its child objects).

Template Name: Frame

UUID

<3D82AB46-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
none			Any

Description

This template defines a frame. Currently the frame can contain objects of the type Mesh and a FrameTransformMatrix.

Optional Data Elements

The following optional data elements are used by Direct3DRM.

FrameTransformMatrix	If this is not present, no local transform is applied to the frame.
Mesh	Any number of mesh objects that become children of the frame. These can be specified inlined or by reference.

Template Name: FloatKeys

UUID

<10DD46A9-775B-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
nValues	DWORD		None
values	array FLOAT	nValues	

Description

This template defines an array of floats and the number of floats in that array. This is used for defining sets of animation keys.

Template Name: TimedFloatKeys

UUID

<F406B180-7B3B-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
time	DWORD		None
tfkeys	FloatKeys		

Description

This template defines a set of floats and a positive time used in animations.

Template Name: AnimationKey

UUID

<10DD46A8-775B-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
keyType	DWORD		None
nKeys	DWORD		
keys	array TimedFloatKeys	nKeys	

Description

This template defines a set of animation keys. The keyType parameter specifies whether the keys are rotation, scale or position keys (using the integers 0, 1 or 2 respectively).

Template Name: AnimationOptions

UUID

<E2BF56C0-840F-11cf-8F52-0040333594A3>

Member Name	Type	Optional Array Size	Optional Data Objects
openclosed	DWORD		None
positionquality	DWORD		

Description

This template allows you to set the D3DRM Animation options. The openclosed parameter can be either 0 for a closed or 1 for an open animation. The positionquality parameter is used to set the position quality for any position keys specified and can either be 0 for spline positions or 1 for linear positions. By default an animation is open and uses linear position keys.

Template Name: Animation

UUID

<3D82AB4F-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
none			any

Description

This template contains animations referencing a previous frame. It should contain one reference to a frame and at least one set of AnimationKeys. It can also contain an AnimationOptions data object.

Optional Data Elements

The following optional data elements are used by Direct3DRM.

AnimationKey	An animation is meaningless without AnimationKeys.
AnimationOptions	If this is not present, then an animation is open and uses linear position keys.

Template Name: AnimationSet

UUID

<3D82AB50-62DA-11cf-AB39-0020AF71E433>

Member Name	Type	Optional Array Size	Optional Data Objects
none			Animation

Description

An AnimationSet contains one or more Animation objects and is the equivalent to the D3D Retained Mode concept of Animation Sets. This means each animation within an animation set has the same time at any given point. Increasing the animation set's time will increase the time for all the animations it contains.

Appendix B: Example

In this appendix we will describe a simple cube and then show how to add textures, frames, and animations to it.

A Simple Cube

This file defines a simple cube that has four red and two green sides. Notice in this file that optional information is being used to add information to the data object defined by the Mesh template.

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;     // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
}
// Define a mesh with 8 vertices and 12 faces (triangles). Use
// optional data objects in the mesh to specify materials, normals
// and texture coordinates.
Mesh CubeMesh {
8;                                  // 8 vertices
1.000000;1.000000;-1.000000;,       // vertex 0
-1.000000;1.000000;-1.000000;,      // vertex 1
-1.000000;1.000000;1.000000;,       // etc...
1.000000;1.000000;1.000000;,
1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;-1.000000;,
-1.000000;-1.000000;1.000000;,
1.000000;-1.000000;1.000000;;

12;                         // 12 faces
3;0,1,2;,                   // face 0 has 3 vertices
3;0,2,3;,                   // etc...
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;

// All required data has been defined. Now define optional data
// using the hierarchical nature of the file format.
MeshMaterialList {
2;                      // Number of materials used
12;                     // A material for each face
0,                      // face 0 uses the first
0,                      // material
0,
0,
0,
0,
0,
0,
1,                      // face 8 uses the second
1,                      // material
```

```
1,
1;;
{RedMaterial}         // References to the definitions
{GreenMaterial}       // of material 0 and 1
}
MeshNormals {
8;                    // define 8 normals
0.333333;0.666667;-0.666667;,
-0.816497;0.408248;-0.408248;,
-0.333333;0.666667;0.666667;,
0.816497;0.408248;0.408248;,
0.666667;-0.666667;-0.333333;,
-0.408248;-0.408248;-0.816497;,
-0.666667;-0.666667;0.333333;,
0.408248;-0.408248;0.816497;;
12;                   // For the 12 faces,
3;0,1,2;,             // define the normals
3;0,2,3;,
3;0,4,5;,
3;0,5,1;,
3;1,5,6;,
3;1,6,2;,
3;2,6,7;,
3;2,7,3;,
3;3,7,4;,
3;3,4,0;,
3;4,7,6;,
3;4,6,5;;
}
MeshTextureCoords {
8;                        // Define texture coords
0.000000;1.000000;        // for each of the vertices
1.000000;1.000000;
0.000000;1.000000;
1.000000;1.000000;
0.000000;0.000000;
1.000000;0.000000;
0.000000;0.000000;
1.000000;0.000000;;
}
}
```

Adding Textures

In order to add textures, we make use of the hierarchical nature of the file format and add an optional **TextureFilename** data object to the **Material** data objects. So, the Material objects now read as follows:

```
Material RedMaterial {
1.000000;0.000000;0.000000;1.000000;;    // R = 1.0, G = 0.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"tex1.ppm";
}
}
Material GreenMaterial {
0.000000;1.000000;0.000000;1.000000;;     // R = 0.0, G = 1.0, B = 0.0
0.000000;
0.000000;0.000000;0.000000;;
0.000000;0.000000;0.000000;;
TextureFilename {
"win95.ppm";
}
}
```

Frames and Animations

This section shows how to add frames and animations to a simple cube.

- Frames
- AnimationSets and Animations

Frames

A frame is expected to take the following structure

```
Frame Aframe {        // The frame name is chosen for convenience.
FrameTransformMatrix {
...transform data...
}
[ Meshes ] and/or [ More frames]
}
```

So, what we're going to do is place the cube mesh we defined earlier inside a frame with an identity transform. We're then going to apply an animation to this frame.

```
Frame CubeFrame {
FrameTransformMatrix {
1.000000, 0.000000, 0.000000, 0.000000,
0.000000, 1.000000, 0.000000, 0.000000,
0.000000, 0.000000, 1.000000, 0.000000,
0.000000, 0.000000, 0.000000, 1.000000;;
}
{CubeMesh}        // We could have the mesh inline, but we'll
                  // use an object reference instead.
}
```

AnimationSets and Animations

Animations and AnimationSets in the file format map directly to Direct3D Retained Mode's animation concepts.

```
Animation Animation0 {        // The name is chosen for convenience.
{ Frame that it applies to - normally a reference }
AnimationKey {
...animation key data...
}
{ ...more animation keys... }
}
```

Animations are then grouped into AnimationSets: AnimationSet AnimationSet0 { // The name is chosen for convenience. { an animation - could be inline or a reference } { ... more animations ... } }

So, what we'll do now is take the cube through an animation

```
AnimationSet AnimationSet0 {
Animation Animation0 {
{CubeFrame}    // Use the frame containing the cube
AnimationKey {
2;             // Position keys
9;             // 9 keys
10; 3; -100.000000, 0.000000, 0.000000;;,
20; 3; -75.000000, 0.000000, 0.000000;;,
30; 3; -50.000000, 0.000000, 0.000000;;,
40; 3; -25.500000, 0.000000, 0.000000;;,
50; 3; 0.000000, 0.000000, 0.000000;;,
60; 3; 25.500000, 0.000000, 0.000000;;,
70; 3; 50.000000, 0.000000, 0.000000;;,
80; 3; 75.500000, 0.000000, 0.000000;;,
90; 3; 100.000000, 0.000000, 0.000000;;;
}
}
}
```

Appendix C: Binary Format

This section details the binary version of the DirectX File Format as introduced with the release of DirectX 3. This appendix should be read in conjunction with the section File Format Architecture.

The binary format is a tokenized representation of the text format. Tokens may be stand-alone or accompanied by primitive data records. Stand-alone tokens give grammatical structure and record-bearing tokens supply the necessary data.

Note that all data is stored in little endian format.

A valid binary data stream consists of a header followed by templates and/or data objects.

This section discusses the following components of the binary file format:

- Header
- Templates
- Data
- Tokens
- Token Records

In addition, example templates are provided, in Example Templates. A binary data object is shown in Example Data.

Header

The following definitions should be used when reading and writing the binary header directly. Note that compressed data streams are not currently supported and are therefore not detailed here.

```
#define XOFFILE_FORMAT_MAGIC \
  ((long)'x' + ((long)'o' << 8) + ((long)'f' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_VERSION \
  ((long)'0' + ((long)'3' << 8) + ((long)'0' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_BINARY \
  ((long)'b' + ((long)'i' << 8) + ((long)'n' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_TEXT   \
  ((long)'t' + ((long)'x' << 8) + ((long)'t' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_COMPRESSED \
  ((long)'c' + ((long)'m' << 8) + ((long)'p' << 16) + ((long)' ' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_32 \
  ((long)'0' + ((long)'0' << 8) + ((long)'3' << 16) + ((long)'2' << 24))

#define XOFFILE_FORMAT_FLOAT_BITS_64 \
  ((long)'0' + ((long)'0' << 8) + ((long)'6' << 16) + ((long)'4' << 24))
```

Templates

A template has the following syntax definition

```
template               : TOKEN_TEMPLATE name TOKEN_OBRACE
                             class_id
                             template_parts
                             TOKEN_CBRACE

template_parts         : template_members_part TOKEN_OBRACKET
                         template_option_info
                         TOKEN_CBRACKET
                       | template_members_list

template_members_part : /* Empty */
                       | template_members_list

template_option_info  : ellipsis
                       | template_option_list

template_members_list :    template_members
                       | template_members_list template_members

template_members      : primitive
                       | array
                       | template_reference

primitive              : primitive_type optional_name TOKEN_SEMICOLON

array                  : TOKEN_ARRAY array_data_type name dimension_list
                         TOKEN_SEMICOLON

template_reference    : name optional_name YT_SEMICOLON

primitive_type         : TOKEN_WORD
                       | TOKEN_DWORD
                       | TOKEN_FLOAT
                       | TOKEN_DOUBLE
                       | TOKEN_CHAR
                       | TOKEN_UCHAR
                       | TOKEN_SWORD
                       | TOKEN_SDWORD
                       | TOKEN_LPSTR
                       | TOKEN_UNICODE
                       | TOKEN_CSTRING

array_data_type        : primitive_type
                       | name

dimension_list         : dimension
                       | dimension_list dimension

dimension              : TOKEN_OBRACKET dimension_size TOKEN_CBRACKET

dimension_size         : TOKEN_INTEGER
                       | name

template_option_list  : template_option_part
                       | template_option_list template_option_part
```

```
template_option_part  : name optional_class_id

name                  : TOKEN_NAME

optional_name         : /* Empty */
                      | name

class_id              : TOKEN_GUID

optional_class_id     : /* Empty */
                      | class_id

ellipsis              : TOKEN_DOT TOKEN_DOT TOKEN_DOT
```

Data

A data object has the following syntax definition

```
object                : identifier optional_name TOKEN_OBRACE
                            optional_class_id
                            data_parts_list
                            TOKEN_CBRACE
data_parts_list       : data_part
                      | data_parts_list data_part

data_part             : data_reference
                      | object
                      | number_list
                      | float_list
                      | string_list

number_list           : TOKEN_INTEGER_LIST

float_list            : TOKEN_FLOAT_LIST

string_list           : string_list_1 list_separator

string_list_1         : string
                      | string_list_1 list_separator string

list_separator        : comma
                      | semicolon

string                : TOKEN_STRING

identifier            : name
                      | primitive_type

data_reference        : TOKEN_OBRACE name optional_class_id TOKEN_CBRACE
```

Tokens

Tokens are written as little endian DWORDs. A list of token values follows. The list is divided into record-bearing and stand-alone tokens.

Record-bearing

```
#define TOKEN_NAME 1
#define TOKEN_STRING 2
#define TOKEN_INTEGER 3
#define TOKEN_GUID 5
#define TOKEN_INTEGER_LIST 6
#define TOKEN_REALNUM_LIST 7
```

Stand-alone

```
#define TOKEN_OBRACE 10
#define TOKEN_CBRACE 11
#define TOKEN_OPAREN 12
#define TOKEN_CPAREN 13
#define TOKEN_OBRACKET 14
#define TOKEN_CBRACKET 15
#define TOKEN_OANGLE 16
#define TOKEN_CANGLE 17
#define TOKEN_DOT 18
#define TOKEN_COMMA 19
#define TOKEN_SEMICOLON 20
#define TOKEN_TEMPLATE 31
#define TOKEN_WORD 40
#define TOKEN_DWORD 41
#define TOKEN_FLOAT 42
#define TOKEN_DOUBLE 43
#define TOKEN_CHAR 44
#define TOKEN_UCHAR 45
#define TOKEN_SWORD 46
#define TOKEN_SDWORD 47
#define TOKEN_VOID 48
#define TOKEN_LPSTR 49
#define TOKEN_UNICODE 50
#define TOKEN_CSTRING 51
#define TOKEN_ARRAY 52
```

Token Records

This section describes the format of the records for each of the record-bearing tokens.

TOKEN_NAME

Field	Type	Size (bytes)	Contents
token	DWORD	4	TOKEN_NAME
count	DWORD	4	Length of name field in bytes
name	BYTE array	count	ASCII name

TOKEN_NAME is a variable length record. The token is followed by a *count* value which specifies the number of bytes which follow in the *name* field. An ASCII name of length *count* completes the record.

TOKEN_STRING

Field	Type	Size (bytes)	Contents
token	DWORD	4	TOKEN_STRING
count	DWORD	4	Length of string field in bytes
string	BYTE array	count	ASCII string
terminator	DWORD	4	TOKEN_SEMICOLON or TOKEN_COMMA

TOKEN_STRING is a variable length record. The token is followed by a *count* value which specifies the number of bytes which follow in the *string* field. An ASCII string of length *count* continues the record which is completed by a terminating token. The choice of terminator is determined by syntax issues discussed elsewhere.

TOKEN_INTEGER

Field	Type	Size (bytes)	Contents
token	DWORD	4	TOKEN_INTEGER
value	DWORD	4	Single integer

TOKEN_INTEGER is a fixed length record. The token is followed by the integer value required.

TOKEN_GUID

Field	Type	Size (bytes)	Contents
token	DWORD	4	TOKEN_GUID
data1	DWORD	4	uuid data field 1
data2	WORD	2	uuid data field 2
data3	WORD	2	uuid data field 3
data4	BYTE array	8	uuid data field 4

TOKEN_GUID is a fixed length record. The token is followed by the four data fields as defined by the OSF DCE standard.

TOKEN_INTEGER_LIST

Field	Type	Size (bytes)	Contents
token	DWORD	4	TOKEN_INTEGER_LIST
count	DWORD	4	Number of integers in list field

list	DWORD array	4 x count	Integer list

TOKEN_INTEGER_LIST is a variable length record. The token is followed by a count value which specifies the number of integers which follow in the *list* field. For efficiency, consecutive integer lists should be compounded into a single list.

TOKEN_REALNUM_LIST

Field	**Type**	**Size (bytes)**	**Contents**
token	DWORD	4	TOKEN_REALNUM_LIST
count	DWORD	4	Number of floats or doubles in list field
list	float/double array	4 or 8 x count	Float or double list

TOKEN_REALNUM_LIST is a variable length record. The token is followed by a count value which specifies the number of floats or doubles which follow in the *list* field. The size of the floating point value (float or double) is determined by the value of *float size* specified in the file header discussed elsewhere. For efficiency, consecutive realnum lists should be compounded into a single list.

Example Templates

Two example binary template definitions are given below. Note that data is stored in little endian format, which is not shown in these illustrative examples.

The closed template *RGB* is identified by the uuid {55b6d780-37ec-11d0-ab39-0020af71e433} and has three members *r*, *g*, and *b* each of type *float*

```
TOKEN_TEMPLATE, TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_OBRACE,
TOKEN_GUID, 55b6d780, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_FLOAT, TOKEN_NAME, 1, 'r', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'g', TOKEN_SEMICOLON,
TOKEN_FLOAT, TOKEN_NAME, 1, 'b', TOKEN_SEMICOLON,
TOKEN_CBRACE
```

The closed template *Matrix4x4* is identified by the uuid {55b6d781-37ec-11d0-ab39-0020af71e433} and has one member, a two-dimensional array named *matrix* of type *float*

```
TOKEN_TEMPLATE, TOKEN_NAME, 9, 'M', 'a', 't', 'r', 'i', 'x', '4', 'x', '4',
TOKEN_OBRACE,
TOKEN_GUID, 55b6d781, 37ec, 11d0, ab, 39, 00, 20, af, 71, e4, 33,
TOKEN_ARRAY, TOKEN_FLOAT, TOKEN_NAME, 6, 'm', 'a', 't', 'r', 'i', 'x',
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_OBRACKET, TOKEN_INTEGER, 4, TOKEN_CBRACKET,
TOKEN_CBRACE
```

Example Data

The binary data object below shows an instance of the *RGB* template defined above. The example object is named *blue* and its three members *r*, *g*, and *b* have the values 0.0, 0.0 and 1.0 respectively. Note that data is stored in little endian format which is not shown in this illustrative example.

```
TOKEN_NAME, 3, 'R', 'G', 'B', TOKEN_NAME, 4, 'b', 'l', 'u', 'e',
TOKEN_OBRACE,
TOKEN_FLOAT_LIST, 3, 0.0, 0.0, 1.0, TOKEN_CBRACE
```

Appendix D: The Conv3ds.exe Utility

The Conv3ds.exe utility program converts 3-D models produced by Autodesk 3-D Studio and other modeling packages into the DirectX file format. By default, it produces binary X files with no templates.

Using Conv3ds.exe

You can run Conv3ds.exe with no options, and it will produce an X file containing a hierarchy of frames. For example, consider the command:

```
conv3ds File.3ds
```

This will produce an X File called File.x. You can use IDirect3DRMFrame::Load to load the frame.

Conv3ds.exe Optional Arguments

Conv3ds.exe -A

If the 3DS file contains key frame data, you can use the **-A** option to produce an X file that contains an animation set. The command for this would be:

```
conv3ds -A File.3ds
```

The File.3ds parameter is the name of the file to be converted. You can use IDirect3DRMAnimationSet::Load to load the animation.

Conv3ds.exe -m

Use the **-m** option to make an X file that contains a single mesh made from all the objects in the 3DS file.

```
conv3ds -m File.3ds
```

Use IDirect3DRMMeshBuilder::Load to load the mesh.

Conv3ds.exe -T

Use the**-T** option to wrap all the objects and frame hierarchies in a single top level frame. Using this option, all the frames and objects in the 3DS file can be loaded with a single call to IDirect3DRMFrame::Load. The first top level frame hierarchy in the X file will be loaded. The frame containing all the other frames and meshes is called "x3ds_filename" (without the .3ds extension). The **-T** option will have no effect if it is used with the **-m** option.

Conv3ds.exe -s

The **-s** option allows you to specify a scale factor for all the objects converted in the 3DS file. For example, the following command makes all objects 10 times bigger:

```
conv3ds -s10 File.3ds
```

The following command makes all objects 10 times smaller:

```
conv3ds -s0.1 File.3ds
```

Conv3ds.exe -r

The **-r** option reverses the winding order of the faces when the 3DS file is converted. If, after converting the 3DS file and viewing it in Direct3D, the object appears "inside-out" try converting it with the **-r** option. All Lightwave models exported as 3DS files need this option. See 3DS Files Produced from Lightwave Objects for details.

Conv3ds.exe -v

The **-v** option turns on verbose output mode. Specify an integer with it. The integers that are currently supported are:

Option	Meaning
-v0	Default. Verbose mode off.
-v1	Prints warnings about bad objects, and prints general information about what the converter is doing.
-v2	Prints basic keyframe information, the objects being included in the conversion process, and information about the objects being saved.
-v3	Very verbose. Mostly useful for the debugging information it provides.

Conv3ds.exe -e

The -e option allows you to change the extension for texture map files. For example, consider the

command:

```
conv3ds -e"ppm" File.3ds
```

If File.3ds contains objects that reference the texture map file brick.gif, the X file will reference the texture map file brick.ppm. The converter does not convert the texture map file. The texture map files must be in the D3DPATH when the resulting X File is loaded.

Conv3ds.exe -x

The **-x** option forces **Conv3ds** to produce a text X file, instead of a binary X file. Text files are larger but can be hand edited easily.

Conv3ds.exe -X

The **-X** option forces **Conv3ds** to include the Direct3DRM X File templates in the file. By default the templates are not included.

Conv3ds.exe -t

The **-t** option specifies that the X File produced will not contain texture information.

Conv3ds.exe -N

The **-N** option specifies that the X file produced will not contain normal vector information. All the Direct3DRM Load calls will generate normal vectors for objects with no normal vectors in the X file.

Conv3ds.exe -c

The **-c** option specifies that the X file produced should not contain texture coordinates. By default, if you use the **-m** option, the mesh that is output will contain (0,0) uv texture coordinates if the 3DS object had no texture coordinates.

Conv3ds.exe -f

The **-f** option specifies that the X file produced should not contain a Frame transformation matrix.

Conv3ds.exe - z and **Conv3ds.exe -Z**

The **-z** and **-Z** options allow you to adjust the alpha face color value of all the materials referenced by objects in the X File. For example, the following command causes **Conv3ds.exe** to add 0.1 to all alpha values under 0.2:

```
conv3ds -z0.1 -Z0.2 File.3ds
```

The following command causes **Conv3ds.exe** to subtract 0.2 from the alpha values for all alphas:

```
conv3ds-z"-0.2" -z1 File.3ds
```

Conv3ds.exe -o

The **-o** option allows you to specify the filename for the .X File produced.

Conv3ds.exe -h

The **-h** option tells the converter not to try to resolve any hierarchy information in the 3DS file (usually produced by the keyframer). Instead, all the objects are output in top level frames if **-m** option is not used.

3DS Files Produced from Lightwave Objects

There are several issues with 3DS files exported by the Trans3d plug-in for Lightwave. These are best handled using the following Conv3ds.exe command:

```
conv3ds -r -N -f -h -T|m trans3dfile.3ds
```

All the 3DS objects produced by Trans3d and the Lightwave object editor need their winding order reversed. Otherwise, they appear "inside-out" when displayed. They contain no surface normal vector information.

Hints and Tips

If you can't see objects produced by Conv3ds.exe after loading them into the Direct3DRM viewer, use the scale option **-s** with a scale factor of approximately 100. This will increase scale of the objects in the X file.

If, after loading the object into the viewer and switching from flat shading into gourard shading the object turns dark grey, try converting with the **-N** option.

If the textures aren't loaded after the object is converted, ensure that the object is referencing either .ppm or .bmp files by using the **-e** option. Also ensure that the textures' widths and heights are a power of 2. Make sure the textures are stored in one of the directories in your D3DPATH.

Currently, Conv3ds.exe can't handle dummy frames used in 3DS animations. It ignores them. However, it will convert any child objects.

DirectInput

This section provides information about the DirectInput® component of DirectX® The information is divided into the following topics:

- About DirectInput
- DirectInput Architecture
- DirectInput Essentials
- DirectInput Tutorials
- DirectInput Reference

About DirectInput

Microsoft® DirectInput® provides support for input devices including the mouse, keyboard, and joystick, as well as for force-feedback (input/output) devices. Like other DirectX® components, DirectInput is based on the Component Object Model (COM).

This release of DirectInput is the first to provide COM-based services for the joystick and other devices such as game pads, flight yokes, and virtual-reality headgear. These new services supersede the Win32® Application Programming Interface (API) functions centered on **joyGetPosEx**, which were previously documented as part of DirectInput.

Also new are the services for the force-feedback devices that are starting to appear in the game market.

Why Use DirectInput?

Aside from providing new services for devices not supported by the Win32 API, such as force feedback game devices, DirectInput gives faster access to input data by communicating directly with the hardware drivers rather than relying on Windows® messages.

The extended services and improved performance of DirectInput make it a valuable tool for games, simulations, and other real-time interactive applications running under Windows.

DirectInput Architecture

This section covers the basic structure of DirectInput and how it works with both the Windows operating system and input hardware. It also introduces the way DirectInput handles device access, input data, and force feedback effects.

Architectural Overview of DirectInput

The basic architecture of DirectInput consists of the DirectInput object, which supports a COM interface, and an object for each input device that provides data. Each device in turn has "object instances," which are individual controls or switches such as keys, buttons, or axes. Individual force feedback effects are also represented by objects.

Note The word "object" is used to describe an entity created by the DirectInput system to support the methods of a COM interface, even when these methods are not being called through an object-oriented programming language such as C++. Somewhat confusingly, "object" may also mean one of the individual controls on an input device.

In the interests of speed and responsiveness, DirectInput works directly with device drivers, bypassing the Windows message system.

DirectInput enables an application to gain access to input devices even when the application is in the background.

The DirectInput Object

The DirectInput object in an application represents the DirectInput subsystem. It is used to enumerate and manage input devices.

You create the DirectInput object by calling the **DirectInputCreate** function, which returns a pointer to an **IDirectInput** COM interface. There are different versions of this interface for the ANSI and Unicode character sets.

Having created the DirectInput object, you can use the methods of the interface to enumerate individual devices available to the system and to create a DirectInputDevice object for each device you wish to use in your application.

The DirectInputDevice Object

Each DirectInputDevice object represents one input device such as a mouse, keyboard, or joystick. (In this documentation, the term "joystick" includes other game devices that behave similarly, such as steering wheels and game pads.)

You create a DirectInputDevice instance by calling the **IDirectInput::CreateDevice** method, which returns a pointer to the **IDirectInputDevice** interface. The **IDirectInputDevice** methods are used to get information about the device, set its properties, and get data from it.

Note A physical device that is really a combination of different types of input devices, such as a keyboard with a trackball, may be represented by two or more DirectInputDevice objects. A force feedback device is represented by a single joystick object that handles both input and output.

DirectInput Device Object Instances

An *object instance*, sometimes called simply an object, is one of the various switches and other controls available on an input device. For example, object instances on a joystick might include the x-axis and y-axis of the stick, several buttons, and a throttle slider. Mouse objects might include two or three buttons, the x-axis and y-axis, and a wheel. For a keyboard, each key is an object.

The application ascertains the number and type of objects available on a device through the **IDirectInputDevice::EnumObjects** method. Individual device objects are not encapsulated as code objects but are described in **DIDEVICEOBJECTINSTANCE** structures.

The DirectInputEffect Object

A DirectInputEffect object represents a force feedback effect that you have defined. It is used to manipulate the effect on the input/output device.

You create an DirectInputEffect object by calling the **IDirectInputDevice2::CreateEffect** function, which returns a pointer to an **IDirectInputEffect** COM interface.

Integration with Windows

Because DirectInput works directly with the device drivers, it either suppresses or ignores mouse and keyboard messages. When using the mouse in exclusive mode, DirectInput suppresses mouse messages; as a result, Windows is unable to show the standard cursor.

DirectInput also ignores mouse and keyboard settings made by the user in Control Panel.

For the keyboard, character repeat settings are not used by DirectInput. When using buffered data, DirectInput interprets each press and release as a single event, with no repetition. When using immediate data, DirectInput is concerned only with the present physical state of the keys, not with keyboard events as interpreted by Windows.

For the mouse, DirectInput ignores Control Panel settings such as acceleration and swapped buttons. Again, DirectInput works directly with the mouse driver, bypassing the subsystem of Windows that interprets mouse data for windowed applications.

Note Settings in the driver itself will be recognized by DirectInput. For example, if the user has a three-button mouse and uses the driver utility software to make the middle button a double-click shortcut, DirectInput will report a click of the middle button as two clicks of the primary button.

For a joystick or other game device, DirectInput does use the calibrations set by the user in Control Panel.

DirectInput Essentials

This section introduces the concepts and components of DirectInput, and provides enough information for you to get started in implementing the DirectInput system in your application.

The following topics are discussed:

- DirectInput Device Enumeration
- DirectInput Devices
- DirectInput Device Data
- Force Feedback
- Designing for Previous Versions of DirectInput

DirectInput Device Enumeration

DirectInput is able to query the system for all available input devices, determine whether they are connected, and return information about them. This process is called enumeration.

If your application is using only the standard keyboard or mouse, or both, you don't need to enumerate the available input devices. As explained under Creating the DirectInput Device, you can simply use predefined global variables when calling the **IDirectInput::CreateDevice** method.

For all other input devices, and for systems with multiple keyboards or mice, you need to call **IDirectInput::EnumDevices** in order to obtain at least the instance GUIDs (globally unique identifiers) so that device objects can be created.

Here's a sample implementation of the **IDirectInput::EnumDevices** method

```
GUID              KeyboardGUID = GUID_SysKeyboard;
// LPDIRECTINPUT  lpdi;   //This has been initialized with
                          //DirectInputCreate and points to
                          //the DirectInput object

lpdi->EnumDevices(DIDEVTYPE_KEYBOARD,
                  DIEnumDevicesProc,
                  &KeyboardGUID,
                  DIEDFL_ATTACHEDONLY);
```

The first parameter determines what types of devices are to be enumerated. It is NULL if you want to enumerate all devices regardless of type; otherwise it is one of the DIDEVTYPE_* values described in the reference for **DIDEVICEINSTANCE**.

The second parameter is a pointer to a callback function that will be called once for each device enumerated. This function can be called by any name; the documentation uses the placeholder name **DIEnumDevicesProc**.

The third parameter to the **EnumDevices** method is any 32-bit value that you want to pass into the callback function. In the examples above, it's a pointer to a variable of type **GUID**, passed in so that the callback can assign a keyboard instance GUID.

The fourth parameter is a flag to request enumeration of either all devices or only those that are attached (DIEDFL_ALLDEVICES or DIEDFL_ATTACHEDONLY).

If your application is using more than one input device, the callback function is a good place to initialize each device as it is enumerated. (For an example, see Tutorial 3: Using the Joystick.) The callback function is where you obtain the instance GUID of the device. You can also perform other processing here, such as looking for particular subtypes of devices.

Here is a sample callback function that checks for the presence of an enhanced keyboard and stops the enumeration as soon as it finds one. It assigns the instance GUID of the last keyboard found to the *KeyboardGUID* variable (passed in as *pvRef* by the **EnumDevices** call above), which can then be used in a call to **IDirectInput::CreateDevice**.

```
BOOL          hasEnhanced;

BOOL CALLBACK DIEnumKbdProc(LPCDIDEVICEINSTANCE lpddi,
                            LPVOID pvRef)
{
  *(GUID*) pvRef = lpddi->guidInstance;
  if (GET_DIDEVICE_SUBTYPE(lpddi->dwDevType) ==
                       DIDEVTYPEKEYBOARD_PCENH)
  {
```

```
    hasEnhanced = TRUE;
    return DIENUM_STOP;
  }
  return DIENUM_CONTINUE;
} // end of callback
```

The first parameter points to a structure containing information about the device. This structure is created for you by DirectInput.

The second parameter points to data passed in from **EnumDevices**. In this case it is a pointer to the variable *KeyboardGUID*. This variable was assigned a default value earlier, but it will be given a new value each time a device is enumerated. It is not actually important what instance GUID you use for a single keyboard, but the code does illustrate a technique for retrieving an instance GUID from the callback.

The return value in this case indicates that enumeration is to stop if the sought-for device has been found, or otherwise that it is to continue. Enumeration will automatically stop as soon as all devices have been enumerated.

DirectInput Devices

This section contains information about the code objects that represent devices such as mice, keyboards, and joysticks. The following topics are covered:

- Device Setup
- Creating a DirectInput Device
- Device Capabilities
- Cooperative Levels
- Device Object Enumeration
- Device Data Formats
- Device Properties
- Acquiring Devices

For information on how to retrieve and interpret data from devices, see DirectInput Device Data.

Device Setup

Your application must obtain a COM interface for each device from which it expects input. It must also prepare each device for use, which requires, at the very least, setting the data format and acquiring the device. You may also wish to carry out other preparatory tasks such as getting information about the devices and changing their properties.

The following tasks are part of the setup process. Certain steps are always required; others may only be necessary if you need further information about devices or need to change default values.

- Create the DirectInput device (required). See Creating a DirectInput Device.
- Get the device capabilities (optional).
- Enumerate the keys, buttons, and axes on the device (optional). See Device Object Enumeration.
- Set the cooperative level (recommended).
- Set the data format (required).
- Set the device properties (optional).
- When ready to read data, acquire the device (required). See Acquiring Devices.

Creating a DirectInput Device

The **IDirectInput::CreateDevice** method is used to obtain a pointer to the **IDirectInputDevice** interface. Methods of this interface are then used to manipulate the device and obtain data.

The following example, where *lpdi* is a pointer to the **IDirectInput** interface, creates a keyboard device:

```
LPDIRECTINPUTDEVICE  lpdiKeyboard;
lpdi->CreateDevice(GUID_SysKeyboard, &lpdiKeyboard, NULL);
```

The first parameter in **IDirectInput::CreateDevice** is an instance GUID that identifies the instance of the device for which the interface is to be created. DirectInput has two predefined GUIDs, *GUID_SysMouse* and *GUID_SysKeyboard*, which represent the system mouse and keyboard, and you can pass these identifiers into the **CreateDevice** function. The global variable *GUID_Joystick* should not be used as a parameter for **CreateDevice**, because it is a product GUID, not an instance GUID.

Note If the workstation has more than one mouse, input from all of them is combined to form the system device. The same is true for multiple keyboards.

For devices other than the system mouse or keyboard, use the instance GUID for the device returned by **IDirectInput::EnumDevices**. The instance GUID for a device will always be the same. You can allow the user to select a device from a list of those enumerated, then save the GUID to a configuration file and use it again in future sessions.

If you want to use the **IDirectInputDevice2** interface methods for force feedback devices, you must obtain a pointer to that interface instead of **IDirectInputDevice**. The following function is a wrapper for the **CreateDevice** method that attempts to obtain the **IDirectInputDevice2** interface. Note the use of macros to call the **Release** and **CreateDevice** methods according to either the C or C++ syntax.

```
HRESULT IDirectInput_CreateDevice2(LPDIRECTINPUT pdi,
                                   REFGUID rguid,
                                   LPDIRECTINPUTDEVICE2 *ppdev2,
                                   LPUNKNOWN punkOuter)
{
    LPDIRECTINPUTDEVICE *pdev;
    HRESULT hres;

    hres = IDirectInput_CreateDevice(pdi, rguid, &pdev, punkOuter);

    if (SUCCEEDED(hres)) {
#ifdef __cplusplus
    hres = pdev->QueryInterface(IID_IDirectInputDevice2,
                                    (LPVOID *)ppdev2);
#else
    hres = pdev->lpVtbl->QueryInterface(pdev,
                        &IID_IDirectInputDevice2,
                               (LPVOID *)ppdev2);
#endif
    IDirectInputDevice_Release(pdev);
    } else {
    *ppdev2 = 0;
    }
    return hres;
}
```

Device Capabilities

Before you begin asking for input from a device, you may need to find out something about its capabilities. Does the joystick have a point-of-view hat? Is the mouse currently attached to the user's machine? Such questions are answered with a call to the **IDirectInputDevice::GetCapabilities** method, which returns the data in a **DIDEVCAPS** structure. As with other such structures in DirectX, you must initialize the **dwSize** member before passing this structure to the function.

Note To optimize speed or memory usage, you can use the smaller **DIDEVCAPS_DX3** structure instead.

Here's an example that checks whether the mouse is attached and whether it has a third axis (presumably a wheel):

```
//  LPDIRECTINPUTDEVICE  lpdiMouse;  // initialized previously

DIDEVCAPS  DIMouseCaps;
HRESULT    hr;
BOOLEAN    HasWheel;

DIMouseCaps.dwSize = sizeof(DIDEVCAPS);
hr = lpdiMouse->GetCapabilities(&DIMouseCaps);
HasWheel = ((DIMouseCaps.dwFlags & DIDC_ATTACHED)
        && (DIMouseCaps.dwAxes > 2));
```

Another way to check for a certain button or axis is to call **IDirectInputDevice::GetObjectInfo** for that object. If the call returns DIERR_OBJECTNOTFOUND, the object is not present. The following code determines whether there is a z-axis even if it is not the third axis.

```
DIDEVICEOBJECTINSTANCE  didoi;

didoi.dwSize = sizeof(DIDEVICEOBJECTINSTANCE);
hr = lpdiMouse->GetObjectInfo(&didoi,DIMOFS_Z, DIPH_BYOFFSET);
HasWheel = SUCCEEDED(hr);
```

Cooperative Levels

The cooperative level of a device determines how the input is shared with other applications and with the Windows system. You set it by using the **IDirectInputDevice::SetCooperativeLevel** method, as in this example:

```
lpdiDevice->SetCooperativeLevel(hwnd,
                                DISCL_NONEXCLUSIVE | DISCL_FOREGROUND)
```

The parameters are the handle of the top-level window associated with the device (generally the application window) and one or more flags. The valid flag combinations are shown in the following table:

Flags	Notes
DISCL_NONEXCLUSIVE | DISCL_BACKGROUND	The default setting
DISCL_NONEXCLUSIVE | DISCL_FOREGROUND	
DISCL_EXCLUSIVE | DISCL_FOREGROUND	Not valid for keyboard
DISCL_EXCLUSIVE | DISC_BACKGROUND	Not valid for keyboard or mouse

Note Although DirectInput provides a default setting, you should still explicitly set the cooperative level, because doing so is the only way to give DirectInput the window handle. Without this handle, DirectInput will not be able to react to situations that involve window messages, such as joystick recalibration.

The cooperative level has two components: whether the device is being used in the foreground or the background, and whether it is being used exclusively or nonexclusively . Both these components require some explanation.

Foreground and Background

A foreground cooperative level means that the input device is available only when the application is in the foreground or, in other words, has the input focus. If the application moves to the background, the device is automatically unacquired, or made unavailable.

A background cooperative level really means "foreground and background." A device with a background cooperative level can be acquired and used by an application at any time.

You will usually want to have foreground access only, since most applications are not interested in input that takes place when another program is in the foreground.

While developing an application, it is useful to have a conditional define that sets the background cooperative level during debugging. This will prevent your application from losing access to the device every time it moves to the background as you switch to a debugging window.

Exclusive and Nonexclusive

The fact that your application is using a device at the exclusive level does not mean that other applications cannot get data from the device. However, it does mean that no other application can also acquire the device exclusively.

Why does it matter? Take the example of a music player that accepts input from a hand-held remote-control device, even when the application is running in the background. Now suppose you run a similar application that plays movies, again in response to signals from the remote control. What happens when the user presses Play? Both programs start playing, which is probably not what the user wants.

To prevent this from happening, each application should have the DISCL_EXCLUSIVE flag set, so that only one of them can be running at a time.

In order to use force feedback effects, an application must have exclusive access to the device.

Windows itself requires exclusive access to the mouse and keyboard. The reason is that mouse and keyboard events such as a click on an inactive window or ALT+TAB could force an application to unacquire the device, with potentially harmful results such as a loss of data from the input buffer.

When an application has exclusive access to the mouse, Windows is not allowed any access at all. No mouse messages are generated. A further side effect is that the cursor disappears.

DirectInput does not allow any application to have exclusive access to the keyboard. If it did, Windows would not have access to the keyboard and the user would not even be able to use CTRL+ALT+DELETE to restart the system.

Device Object Enumeration

It may be necessary for your application to determine what buttons or axes are available on a given device. To do this you enumerate the device objects in much the same way you enumerate devices.

To some extent **IDirectInputDevice::EnumObjects** overlaps the functionality of **IDirectInputDevice::GetCapabilities**. Either method may be used to determine how many buttons or axes are available. However, **EnumObjects** is really intended for cataloguing all the available objects rather than checking for a particular one. The DIQuick application in the DirectX code samples in the Platform SDK References, for example, uses **EnumObjects** to populate the list on the Objects page for the selected device.

Like **IDirectInput::EnumDevices**, the **EnumObjects** function has a callback function that gives you the chance to do other processing on each object - for example, adding it to a list or creating a corresponding element on a user interface.

Here's a callback function that simply extracts the name of each object so that it can be added to a string list or array. This standard callback is documented under the placeholder name **DIEnumDeviceObjectsProc**, but you can give it any name you like. Remember, this function is called once for each object enumerated.

```
char  szName[MAX_PATH];

BOOL CALLBACK DIEnumDeviceObjectsProc(
                      LPCDIDEVICEOBJECTINSTANCE lpddoi,
                      LPVOID pvRef)
{
  lstrcpy(szName, lpddoi->tszName);
  // Now add szName to a list or array
  .
  .
  .
  return DIENUM_CONTINUE;
}
```

The first parameter points to a structure containing information about the object. This structure is created for you by DirectInput.

The second parameter is an application-defined pointer to data, equivalent to the second parameter to **EnumObjects**. In the example, this parameter is not used.

The return value in this case indicates that enumeration is to continue as long as there are still objects to be enumerated.

Now here's the call to the **EnumObjects** method, which puts the callback function to work.

```
lpdiMouse->EnumObjects(DIEnumDeviceObjectsProc,
                       NULL, DIDFT_ALL);
```

The first parameter is the address of the callback function.

The second parameter can be a pointer to any data you want to use or modify in the callback. The example does not use this parameter and so passes NULL.

The third parameter is a flag to indicate which type or types of objects are to be included in the enumeration. In the example, all objects are to be enumerated. To restrict the enumeration, you can use one or more of the other DIDFT_* flags listed in the reference for **IDirectInput::EnumDevices**.

Note Some of the DIDFT_* flags are combinations of others; for example, DIDFT_AXIS is equivalent to DIDFT_ABSAXIS | DIDFT_RELAXIS.

Device Data Formats

Setting the data format for a device is an essential step before you can acquire and begin using the device. The **IDirectInputDevice::SetDataFormat** method tells DirectInput what device objects will be used and how the data will be arranged.

The examples in the reference for the **DIDATAFORMAT** structure and **DIOBJECTDATAFORMAT** structure will give you an idea of how to set up custom data formats for nonstandard devices. Fortunately, this step is not necessary for the joystick, keyboard, and mouse. DirectInput provides four global variables, *c_dfDIJoystick*, *c_dfdiJoystick2*, *c_dfDIKeyboard*, and *c_dfDIMouse*, which can be passed into **SetDataFormat** to create a standard data format for these devices.

Here is an example, where *lpdiMouse* is an initialized pointer to the mouse device object:

```
lpdiMouse->SetDataFormat(&c_dfDIMouse);
```

Note You cannot change the **dwFlags** member in the predefined **DIDATAFORMAT** global variables (for example, in order to change the property of an axis), because they are **const** variables. To change properties, use the **IDirectInputDevice::SetProperty** method after setting the data format but before acquiring the device.

Device Properties

Properties of DirectInput devices include the size of the data buffer, the range and granularity of values returned from an axis, whether axis data is relative or absolute, and the dead zone and saturation values for a joystick axis, which affect the relationship between the physical position of the stick and the reported data. Specialized devices may have other properties as well.

With one exception – the gain property of a force feedback device – properties may only be changed when the device is in an unacquired state.

Before calling the **IDirectInputDevice::SetProperty** or **IDirectInputDevice::GetProperty** methods you need to set up a property structure, which consists of a **DIPROPHEADER** structure and one or more elements for data. There are potentially a great variety of properties for input devices, and **SetProperty** must be able to work with all sorts of structures defining those properties. The purpose of the **DIPROPHEADER** structure is to define the size of the property structure and how the data is to be interpreted.

DirectInput includes two predefined property structures:

- **DIPROPDWORD** defines a structure containing a **DIPROPHEADER** and a **DWORD** data member, for properties that require a single value, such as a buffer size.
- **DIPROPRANGE** is for range properties, which require two values (maximum and minimum). It consists of a **DIPROPHEADER** and two **LONG** data members.

For **SetProperty**, the data members of the property structure are the values you want to set. For **GetProperty**, the current value is returned in these members.

Before the call to **GetProperty** or **SetProperty**, the **DIPROPHEADER** structure must be initialized with the following:

- The size of the property structure
- The size of the **DIPROPHEADER** structure itself
- An object identifier
- A "how" code indicating the way the object identifier should be interpreted

When getting or setting properties for a whole device, the object identifier *dwObj* is zero and the "how" code *dwHow* is DIPH_DEVICE. If you want to get or set properties for a device object (for example, a particular axis), the combination of *dwObj* and *dwHow* values identifies the object. For details, see the reference for the **DIPROPHEADER** structure.

After setting up the property structure, you pass the address of its header into **GetProperty** or **SetProperty**, along with an identifier for the property you want to obtain or change.

The following values are used to identify the property passed to **SetProperty** and **GetProperty.** See the reference for **IDirectInputDevice::GetProperty** for more information.

- DIPROP_BUFFERSIZE. See also Buffered and Immediate Data.
- DIPROP_AXISMODE. See also Relative and Absolute Axis Coordinates.
- DIPROP_CALIBRATIONMODE
- DIPROP_GRANULARITY
- DIPROP_FFGAIN
- DIPROP_FFLOAD
- DIPROP_AUTOCENTER
- DIPROP_RANGE
- DIPROP_DEADZONE
- DIPROP_SATURATION

For more information about the last three properties, see also Interpreting Joystick Axis Data.

The following example sets the buffer size for a device so that it will hold 10 data items:

```
DIPROPDWORD  dipdw;
HRESULT      hres;
dipdw.diph.dwSize = sizeof(DIPROPDWORD);
dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
dipdw.diph.dwObj = 0;
dipdw.diph.dwHow = DIPH_DEVICE;
dipdw.dwData = 10;
hres = lpdiDevice->SetProperty(DIPROP_BUFFERSIZE, &dipdw.diph);
```

Acquiring Devices

Acquiring a DirectInput device means giving your application access to it. As long as a device is acquired, DirectInput is making its data available to your application. If the device is not acquired, you may manipulate its characteristics but not obtain any data.

Acquisition is not permanent; your application may acquire and unacquire a device many times.

In certain cases, depending on the cooperative level, a device may be unacquired automatically whenever your application moves to the background. The mouse is automatically unacquired when the user clicks on a menu, because at this point Windows takes over the device.

You need to unacquire a device before changing its properties. The only exception is that you may change the gain for a force feedback device while it is in an acquired state.

Why is the acquisition mechanism needed? There are two main reasons.

First, DirectInput has to be able to tell the application when the flow of data from the device has been interrupted by the system. For instance, if the user has switched to another application with ALT+TAB, and used the input device in that application, your application needs to know that the input no longer belongs to it and that the state of the buffers may have changed. Or consider an application with the DISCL_FOREGROUND cooperative level. The user presses the SHIFT key, and while continuing to press it switches to another application. Then the user releases the key and switches back to the first application. As far as the first application is concerned, the SHIFT key is still down. The acquisition mechanism, by telling the application that input was lost, allows it to recover from these conditions.

Second, because your application can alter the properties of the device, without safeguards DirectInput would have to check the properties each time you wanted to get data with the **IDirectInputDevice::GetDeviceState** or **IDirectInputDevice::GetDeviceData** methods. Obviously this would be very inefficient. Even worse, potentially disastrous things could happen like a hardware interrupt accessing a data buffer just as you were changing the buffer size. So DirectInput requires your application to unacquire the device before changing properties. When you reacquire it, DirectInput looks at the properties and decides on the optimal way of transferring data from the device to the application. This is done only once, thereby making **GetDeviceState** and **GetDeviceData** very fast.

Since the most common cause of losing a device is that your application moves to the background, you may want to reacquire devices whenever your application is activated. However, this mechanism is not going to cover all cases where a device is unacquired, especially for devices other than the standard mouse or keyboard. Because your application may unacquire a device unexpectedly, you need to have a mechanism for checking the acquisition state before attempting to get data from the device. The Scrawl sample application does this in the **Scrawl_OnMouseInput** function, where a DIERR_INPUTLOST error triggers a message to reacquire the mouse. (See also Tutorial 2: Using the Mouse.)

There's no harm in attempting to reacquire a device that is already acquired. Redundant calls to **IDirectInputDevice::Acquire** are ignored, and the device can always be unacquired with a single call to **IDirectInputDevice::Unacquire**.

Remember, Windows doesn't have access to the mouse when your application is using it in exclusive mode. If you want to let Windows have the mouse, you must let it go. There's an example in Scrawl, which responds to a click of the right button by unacquiring the mouse, putting the Windows cursor in the same spot as its own, popping up a context menu, and letting Windows handle the input until a menu choice is made.

DirectInput Device Data

This section covers the basic concepts of getting data from DirectInput devices.

- Buffered and Immediate Data
- Time Stamps and Sequence Numbers
- Polling and Events
- Relative and Absolute Axis Coordinates

Specific details about mouse, keyboard, and joystick data are given in the following sections:

- Mouse Data
- Keyboard Data
- Joystick Data

Examples of retrieving data from input devices are found in the following tutorials:

- Tutorial 1: Using the Keyboard
- Tutorial 2: Using the Mouse
- Tutorial 3: Using the Joystick

Buffered and Immediate Data

DirectInput supplies two types of data: buffered and immediate. Buffered data is a record of events that is stored until an application retrieves it. Immediate data is a snapshot of the current state of a device.

You might use immediate data in an application that is concerned only with the current state of a device: for example, a flight combat simulation that responds to the current position of the joystick and a pressed "fire" button. Buffered data might be the better choice where events are more important than states: for example, in an application that responds to movement of the mouse and button clicks.

You get immediate data with the **IDirectInputDevice::GetDeviceState** method. As the name implies, this method simply returns the current state of the device: for example, whether each button is up or down. The method provides no data about what has happened with the device since the last call, apart from implicit information you can derive by comparing the current state with the last one. If the user manages to press and release a button between two calls to **GetDeviceState**, your application won't know anything about it. On the other hand, if the user is holding a button down, **GetDeviceState** will continue reporting "button down" until the user releases it.

This way of reporting the device state is different from the way Windows reports events with one-time messages like WM_LBUTTONDOWN; it is more akin to the results from the Win32 **GetKeyboardState** function. If you are polling a device with **GetDeviceState**, you are responsible for determining what constitutes a button click, a double-click, a single keystroke, and so on, and for ensuring that your application doesn't keep responding to a button-down or key-down state when it's not appropriate to do so.

With buffered data, events are stored until you're ready to deal with them. Every time a button or key is pressed or an axis is moved, information about the event is placed in a **DIDEVICEOBJECTDATA** structure in the buffer. If the buffer overflows, new data is lost. Your application reads the buffer with a call to **IDirectInputDevice::GetDeviceData**. You can read any number of items at a time.

Reading an item normally deletes it from the buffer, but you also have the choice of peeking without deleting. Peeking could be used, for example, in an application that treats a double-click differently from a single click. Before responding to the first click, the application would peek at the buffer to see if another button-down event took place within a certain time interval after the first.

In order to get buffered data you must first set the buffer size with the **IDirectInputDevice::SetProperty** method. (See the example under Device Properties.) You set the buffer size before acquiring the device for the first time. For reasons of efficiency, the default size of the buffer is zero. You will not be able to obtain buffered data unless you change this value.

Note The size of the buffer is measured in items of data for that type of device, not in bytes or words.

You should check the value of the *pdwInOut* parameter after a call to the **GetDeviceData** method. The number of items actually retrieved from the buffer is returned in this variable.

The DIQuick application supplied with the DirectX code samples in the Platform SDK Reference lets you see both immediate and buffered data from a device. After you create the device in the application window, set its properties on the Mode page. Now, on the Data page, you see immediate data on the left and buffered data on the right.

Note For devices that do not generate interrupts, such as analog joysticks, DirectInput does not obtain any data until you call the **IDirectInputDevice2::Poll** method. For more information, see Polling and Events.

For examples of retrieving buffered data, see **IDirectInputDevice::GetDeviceData**.

See also:

- Time Stamps and Sequence Numbers

- Mouse Data
- Keyboard Data
- Joystick Data

Time Stamps and Sequence Numbers

When DirectInput input data is buffered (see Buffered and Immediate Data), each **DIDEVICEOBJECTDATA** structure contains not only information about the type of event and the device object associated with it, but also a time stamp and a sequence number.

The **dwTimeStamp** member contains the system time in milliseconds at the time the event took place. This is equivalent to the value that would have been returned by the Win32 **GetTickCount** function.

The **dwSequence** member contains a sequence number assigned by DirectInput. The DirectInput system keeps a single sequence counter, which is incremented by each non-simultaneous buffered event from any device. You can use this number to compare events from different devices and see which came first. The **DISEQUENCE_COMPARE** macro takes wraparound into account.

Simultaneous events are assigned the same sequence number. If a mouse or joystick is moved diagonally, for example, the changes in the x-axis and the y-axis have the same sequence number.

Note Events are always placed in the buffer in chronological order, so you don't need to check the sequence numbers just to sort the events from a single device.

Polling and Events

There are two ways to find out whether input data is available: by polling and by event notification.

Polling a device means regularly getting the current state of the device objects with **IDirectInputDevice::GetDeviceState** or retrieving the contents of the buffer with **IDirectInputDevice::GetDeviceData**. Polling is typically used by real-time games that are never idle but are constantly updating and rendering the game world.

Event notification is suitable for applications like the Scrawl sample that wait for input before doing anything. To use event notification, you set up a thread synchronization object with the Win32 **CreateEvent** function and then associate this event with the device by passing its handle to the **IDirectInputDevice::SetEventNotification** method. The event is then signaled by DirectInput whenever the state of the device changes. Your application can receive notification of the event with a Win32 function such as **WaitForSingleObject**, and then respond by checking the input buffer to find out what the event was. For sample code, see the Scrawl sample and the reference for **IDirectInputDevice::SetEventNotification**.

Some joysticks and other game devices, or particular objects on them, do not generate hardware interrupts and will not return any data or signal any events until you call the **IDirectInputDevice2::Poll** method. To find out whether this is necessary, first set the data format for the device, then call the **IDirectInputDevice::GetCapabilities** method and check for the DIDC_POLLEDDATAFORMAT flag in the **DIDEVCAPS** structure.

Do not confuse the DIDC_POLLEDDATAFORMAT flag with the DIDC_POLLEDDEVICE flag. The latter will be set if any object on the device requires polling. You can then find out whether this is the case for a particular object by calling the **IDirectInputDevice::GetObjectInfo** method and checking for the DIDOI_POLLED flag in the **DIDEVICEOBJECTINSTANCE** structure.

The DIDC_POLLEDDEVICE flag describes the worst case for the device, not the actual situation. For example, HID keyboards will be marked as DIDC_POLLEDDEVICE because the LEDs that indicate status, such as CAPS LOCK, require polling. However, the standard keyboard data format does not read the LEDs, so DIDC_POLLEDDATAFORMAT will not be set. Polling the device under these conditions is pointless, because the device objects that require polling (the LEDs) are inaccessible from the data format anyway.

It doesn't hurt to call the **IDirectInputDevice2::Poll** method for any input device. If the call is unnecessary, it will have no effect and will be very fast.

Relative and Absolute Axis Coordinates

Axis coordinates may be returned as relative values; that is, the amount by which they have changed since the last call to the **IDirectInputDevice::GetDeviceState** method or, in the case of buffered input, since the last item was put in the buffer.

Absolute axis coordinates are a running total of all the relative coordinates returned by the system since the device was acquired; in other words, they show the position of the axis in relation to a fixed point.

By default, mouse axes are reported as relative coordinates and joystick axes as absolute coordinates. You can use the **IDirectInputDevice::SetProperty** method to change the default behavior of any axis or all the axes of a device.

Mouse Data

To set up the mouse device for data retrieval, first call the **IDirectInputDevice::SetDataFormat** method with the *c_dfDIMouse* global variable as the parameter.

For maximum performance in a full-screen application, set the cooperative level to DISCL_EXCLUSIVE | DISCL_FOREGROUND. Note that the exclusive setting will cause the Windows cursor to disappear. Remember too that the DISCL_FOREGROUND setting will cause the application to lose access to the mouse when you switch to a debugging window. Changing to DISCL_BACKGROUND will allow you to debug the application more easily, at a cost in performance.

The following sections give more information about getting and interpreting immediate and buffered mouse data.

See also:

- Device Data Formats
- Cooperative Levels

Immediate Mouse Data

To retrieve the current state of the mouse, call **IDirectInputDevice::GetDeviceState** with a pointer to a **DIMOUSESTATE** structure. The mouse state returned in the structure includes axis data and the state of each of the buttons.

The first three members of the **DIMOUSESTATE** structure hold the axis coordinates. (See Interpreting Mouse Axis Data.)

The **rgbButtons** member is an array of bytes, one for each of four buttons. Generally the first element in the array is the left button, the second is the right button, the third is the middle button, and the fourth is any other button. The high bit is set if the button is down and clear if the button is up or not present.

See also Buffered and Immediate Data.

Buffered Mouse Data

To retrieve buffered data from the mouse, you must first set the buffer size (see Device Properties). The default size of the buffer is zero, so this step is essential. You then declare an array of **DIDEVICEOBJECTDATA** structures with the same number of elements as the buffer size.

After acquiring the device, you can examine and flush the buffer anytime by using the **IDirectInputDevice::GetDeviceData**method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the mouse. For instance, a typical mouse contains four objects or input sources: x-axis, y-axis, button 0 and button 1. If the user presses button 0 and moves the mouse diagonally, the array passed to **IDirectInputDevice::GetDeviceData** will have three elements filled in: an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis.

You can determine which object an element in the array refers to by checking the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure against the following values:

- DIMOFS_BUTTON0 *to* DIMOFS_BUTTON3
- DIMOFS_X
- DIMOFS_Y
- DIMOFS_Z

Each of these values is derived from the offset of the data for the object in a **DIMOUSESTATE** structure. For example, DIMOFS_BUTTON0 is equivalent to the offset of **rgbButtons[0]** in the **DIMOUSESTATE** structure. With the macros you can use simple comparisons to determine which device object is associated with an item in the buffer. For example:

```
DIDEVICEOBJECTDATA  *lpdidod;
int                 n;
.
.
.
/* MouseBuffer is an array of DIDEVICEOBJECTDATA structures
   that has been set by a call to GetDeviceData.
   n is incremented in a loop that examines all filled elements
   in the array. */
lpdidod = &MouseBuffer[n];
if (( (int) lpdidod->dwOfs == DIMOFS_BUTTON0)
      && (lpdidod->dwData & 0x80))
{
 ;  // do something in response to left button press
}
```

The data for the change of state of the device object is located in the **dwData** member of the **DIDEVICEOBJECTDATA** structure. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **dwData** is significant; the high bit of this byte is set if the button was pressed and clear if the button was released. In other words, the button was pressed if (**dwData** & 0x80) is non-zero.

For more information on the other members of the **DIDEVICEOBJECTDATA** structure, see Time Stamps and Sequence Numbers.

Interpreting Mouse Axis Data

The data returned for the x-axis and y-axis of a mouse indicates the movement of the mouse itself, not the cursor. The units of measurement, sometimes called mickeys, are based on the actual values returned by the mouse hardware. Because DirectInput communicates directly with the mouse driver, the values for mouse speed and acceleration set by the user in Control Panel do not affect this data.

Axis data returned from the mouse can be either relative or absolute. (See Relative and Absolute Axis Coordinates.) Because a mouse is a relative device - unlike a joystick, it does not have a home position - relative data is returned by default.

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to absolute, call **IDirectInputDevice::SetProperty** with the DIPROP_AXISMODE value in the *rguid* parameter and with DIPROPAXISMODE_ABS in the **dwData** member of the **DIPROPDWORD** structure.

When the axis mode for the mouse is set to relative, the axis coordinate represents the number of mickeys that the device has been moved along the axis since the last value was returned. A negative value indicates that the mouse was moved to the left for the x-axis, or away from the user for the y-axis, or that the z-axis (the wheel) was rotated back. Positive values indicate movement in the opposite direction.

When the axis mode is set to absolute, the coordinates are simply a running total of all relative motions received by DirectInput. The axis coordinates are not initialized to any particular value when the device is acquired, so your application should treat absolute values as relative to an unknown origin. You can record the current absolute position whenever the device is acquired and save it as the "virtual origin." This virtual origin can then be subtracted from subsequent absolute coordinates retrieved from the device to compute the relative distance the mouse has moved from the point of acquisition.

The data returned for the axis coordinates is also affected by the granularity property of the device. For the x-axis and y-axis of the mouse, granularity is normally 1, meaning that the minimum change in value is 1. For the wheel axis it may be larger.

Checking for Lost Mouse Input

Because Windows may force your application to unacquire the mouse when you have set the cooperative level to DISCL_FOREGROUND and the focus switches to another application or even to the menu in your own application, you should check for the DIERR_INPUTLOST return value from the **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState** methods, and attempt to reacquire the mouse if necessary. (See Acquiring Devices.)

Note You should not attempt to reacquire the mouse on getting a DIERR_NOTACQUIRED error. If you do, you are likely to get caught in an infinite loop: acquisition will fail, you will get another DIERR_NOTACQUIRED error, and so on.

Keyboard Data

As far as DirectInput is concerned, the keyboard is not a text input device but a game pad with many buttons. When your application requires text input, don't use DirectInput methods; it is far easier to retrieve the data from the normal Windows messages, where you can take advantage of services such as character repeat and translation of physical keys to virtual keys. This is particularly important for languages other than English, which may require special translation of key presses.

To set up the keyboard device for data retrieval, you must first call the **IDirectInputDevice::SetDataFormat** method with the *c_dfDIKeyboard* global variable as the parameter. (See Device Data Formats.)

The following sections give more information about obtaining and interpreting keyboard data.

Immediate Keyboard Data

To retrieve the current state of the keyboard, call the **IDirectInputDevice::GetDeviceState** method with a pointer to an array of 256 bytes that will hold the returned data.

The **GetDeviceState** method behaves in the same way as the Win32 **GetKeyboardState** function, returning a snapshot of the current state of the keyboard. Each key is represented by a byte in the array of 256 bytes whose address was passed as the *lpvData* parameter. If the high bit of the byte is set, the key is down. The array is most conveniently indexed with the DirectInput Keyboard Device Constants. (See also Interpreting Keyboard Data.)

Here is an example that does something in response to the ESC key being down:

```
// LPDIRECTINPUTDEVICE  lpdiKeyboard;  // previously initialized
                                       // and acquired

BYTE  diKeys[256];
if (lpdiKeyboard->GetDeviceState(256, diKeys) == DI_OK)
{
  if (diKeys[DIK_ESCAPE] & 0x80) DoSomething();
}
```

Buffered Keyboard Data

To retrieve buffered data from the keyboard, you must first set the buffer size (see Device Properties). This step is essential, because the default size of the buffer is zero. You then declare an array of **DIDEVICEOBJECTDATA** structures with the same number of elements as the buffer size.

After acquiring the keyboard device, you can examine and flush the buffer anytime by using the **IDirectInputDevice::GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single key; that is, a press or release. Because DirectInput gets the data directly from the keyboard, any settings for character repeat in Control Panel are ignored. This means that a keystroke is only counted once, no matter how long the key is held down.

You can determine which key an element in the array refers to by checking the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure against the DirectInput Keyboard Device Constants. (See also Interpreting Keyboard Data.)

The data for the change of state of the key is located in the **dwData** member of the **DIDEVICEOBJECTDATA** structure. Only the low byte of **dwData** is significant; the high bit of this byte is set if the key was pressed and clear if it was released. In other words, the key was pressed if (**dwData** & 0x80) is non-zero.

Interpreting Keyboard Data

This section covers the identification of keys for which data is reported by the **IDirectInputDevice:GetDeviceState** and **IDirectInputDevice:GetDeviceData** methods. For more information on interpreting the data from **GetDeviceData**, see Time Stamps and Sequence Numbers.

In one important respect, DirectInput differs from other ways of reading the keyboard in Windows. Keyboard data refers not to virtual keys but to the actual physical keys; that is, the scan codes. DIK_ENTER, for example, refers to the ENTER key on the main keyboard but not to the one on the numerical keypad. (For a list of the DIK_* values, see Keyboard Device Constants.)

DirectInput defines a constant for each key on the enhanced keyboard as well as the additional keys found on international keyboards. Because NEC keyboards support different scan codes than the PC enhanced keyboards, DirectInput translates NEC key scan codes into PC enhanced scan codes where possible.

Not all PC enhanced keyboards have the Windows keys (DIK_LWIN, DIK_RWIN, and DIK_APPS). There is no way to determine whether the keys are physically available.

There is no key code for the PAUSE key. The PC enhanced keyboard does not generate a separate scan code for this key; rather, it synthesizes a "pause" from the DIK_LCONTROL and DIK_NUMLOCK codes.

Laptops and other small computers often do not implement a full set of keys. Instead, some keys (typically numeric keypad keys) are multiplexed with other keys, selected by an auxiliary mode key, which does not generate a separate scan code.

If the keyboard subtype indicates a PC XT or PC AT keyboard, then the following keys are not available: DIK_F11, DIK_F12, and all the extended keys (DIK_* values greater than 0x7F). Furthermore, the PC XT keyboard lacks DIK_SYSRQ.

Japanese keyboards, particularly the NEC PC-98 keyboards, contain a substantially different set of keys than U.S. keyboards. For more information, see DirectInput and Japanese Keyboards.

Checking for Lost Keyboard Input

Because Windows may force your application to unacquire the keyboard when you have set the cooperative level to DISCL_FOREGROUND and the focus switches to another application, you should check for the DIERR_INPUTLOST return value from the **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState** methods, and attempt to reacquire the keyboard if necessary. (See Acquiring Devices.)

Note You should not attempt to reacquire the keyboard on getting a DIERR_NOTACQUIRED error. If you do, you are likely to get caught in an infinite loop: acquisition will fail, you will get another DIERR_NOTACQUIRED error, and so on.

Joystick Data

The following sections cover getting and interpreting data from a joystick or other similar input device such as a game pad or flight yoke.

To set up the joystick device for data retrieval, first call the **IDirectInputDevice::SetDataFormat** method with the *c_dfDIJoystick* or *c_dfDIJoystick2* global variable as the parameter. (See Device Data Formats.)

Because some device drivers do not notify DirectInput of changes in state until explicitly asked to do so, you should always call the **IDirectInputDevice2::Poll** method before attempting to retrieve data from the joystick. For more information, see Polling and Events.

Immediate Joystick Data

To retrieve the current state of the joystick, call the **IDirectInputDevice::GetDeviceState** method with a pointer to a **DIJOYSTATE** or **DIJOYSTATE2** structure, depending on whether the data format was set with *c_dfDIJoystick* or *c_dfDIJoystick2*. (See Device Data Formats.) The joystick state returned in the structure includes the coordinates of the axes, the state of the buttons, and the state of the point-of-view controllers.

The first seven members of the **DIJOYSTATE** structure hold the axis coordinates. The last of these, **rglSlider**, is an array of two values. (See Interpreting Joystick Axis Data.)

The **rgdwPOV** member contains the position of up to four point-of-view controllers in hundredths of degrees clockwise from north (or forward). The center position is reported as -1. For controllers that have only five positions, **dwPOV** will be one of the following values:

-1

0

90 * DI_DEGREES

180 * DI_DEGREES

270 * DI_DEGREES.

Note Some drivers report a value of 65,535 instead of -1 for the neutral position. You should check for a centered POV indicator as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

The **rgbButtons** member is an array of bytes, one for each of 32 or 128 buttons, depending on the data format. For each button, the high bit is set if the button is down and clear if the button is up or not present.

The **DIJOYSTATE2** structure has additional members for information about the velocity, acceleration, force, and torque of the axes.

See also Buffered and Immediate Data.

Buffered Joystick Data

To retrieve buffered data from the joystick, you must first set the buffer size (see Device Properties) and declare an array of **DIDEVICEOBJECTDATA** structures. The number of elements required in this array is the same as the buffer size. After acquiring the device, you can examine and flush the buffer anytime with the **IDirectInputDevice:GetDeviceData** method. (See Buffered and Immediate Data.)

Each element in the **DIDEVICEOBJECTDATA** array represents a change in state for a single object on the joystick. For instance, a simple joystick contains four objects or input sources: x-axis, y-axis, button 0 and button 1. If the user presses button 0 and moves the stick diagonally, the array passed to **IDirectInputDevice::GetDeviceData** will have three elements filled in: an element for button 0 being pressed, an element for the change in the x-axis, and an element for the change in the y-axis.

You can determine which object an element in the array refers to by checking the **dwOfs** member of the **DIDEVICEOBJECTDATA** structure against the following values:

- DIJOFS_X
- DIJOFS_Y
- DIJOFS_Z
- DIJOFS_Rx
- DIJOFS_Ry
- DIJOFS_Rz
- DIJOFS_BUTTON0 *to* DIJOFS_BUTTON31 *or* DIJOFS_BUTTON(*n*)
- DIJOFS_POV(*n*)
- DIJOFS_SLIDER(*n*)

Each of these values is equivalent to the offset of the data for the object in a **DIJOYSTATE** structure. For example, DIJOFS_BUTTON0 is equivalent to the offset of **rgbButtons[0]** in the **DIJOYSTATE** structure. You can use simple comparisons to determine which device object is associated with an item in the buffer. For example:

```
DIDEVICEOBJECTDATA  *lpdidod;
int                 n;
.
.
.
/* JoyBuffer is an array of DIDEVICEOBJECTDATA structures
   that has been set by a call to GetDeviceData.
   n is incremented in a loop that examines all filled elements
   in the array. */
lpdidod = &JoyBuffer[n];
if (( (int) lpdidod->dwOfs == DIJOFS_BUTTON0)
   && (lpdidod->dwData & 0x80))
{
 ;  // do something in response to press of primary button
}
```

The data for the change of state of the device object is located in the **dwData** member of the **DIDEVICEOBJECTDATA** structure. For axes, the coordinate value is returned in this member. For button objects, only the low byte of **dwData** is significant; the high bit of this byte is set if the button is pressed and clear if the button is released.

For the other members, see Time Stamps and Sequence Numbers.

Interpreting Joystick Axis Data

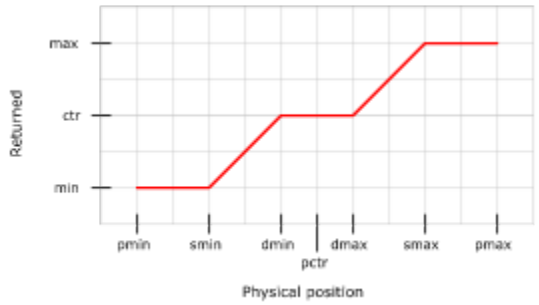
Axis values for the joystick are like those for the mouse: the value returned for the x-axis is greater as the stick moves to the right, and the value for the y-axis increases as the stick moves toward the user.

Data is in arbitrary units determined by the range property of the axis. For example, if the range for the stick's x-axis is 0 to 10,000, a unit is one ten-thousandth of the stick's left-right travel, and the center position is 5,000. For some axes the granularity property may be greater than 1, in which case values will be rounded off; for example, if the granularity is 10 then values will be reported as 0, 10, 20, and so on.

Axis data is also affected by the dead zone, a region around the center position in which motion is ignored. The dead zone provides tolerance for a slight deviation from the true center position for either or both axes of the stick. An axis value within the range of the dead zone is reported as true center.

The saturation property of an axis is a zone of tolerance at the minimum and maximum of the range. An axis value within this zone is reported as the minimum or maximum value. The purpose of the saturation property is to allow for slight differences between, for example, the minimum x-axis value reported at the top left and bottom left positions of the stick.

The following diagram shows the effect of the dead zone and the saturation zones. The vertical axis represents the returned axis values, where *min* and *max* are the lower and upper limits of the reported range and *ctr* is the reported center. The horizontal axis shows the physical position of the stick, where *pmin* and *pmax* are the extremes of the physical range, *pctr* is neutral position of the axis, *dmin* and *dmax* are the limits of the dead zone, and *smin* and *smax* are the boundaries of the lower and upper saturation zones. The lower saturation zone lies between *pmin* and *smin*; the upper saturation zone lies between *smax* and *pmax*; and the dead zone lies between *dmin* and *dmax*.



For more information on joystick properties, see the following:

- **IDirectInputDevice::GetProperty**
- **IDirectInputDevice::SetProperty**
- **DIPROPRANGE**

Axis coordinates from the joystick can be either relative or absolute. (See Relative and Absolute Axis Coordinates.) Because a joystick is an absolute device-unlike a mouse, it cannot travel infinitely far along any axis-absolute data is returned by default.

The axis mode, which specifies whether relative or absolute data should be returned, is a property that can be changed before the device is acquired. (See Device Properties.) To set the axis mode to relative, call the **IDirectInputDevice::SetProperty** method with the DIPROP_AXISMODE value in the *rguid* parameter and with DIPROPAXISMODE_REL in the **dwData** member of the **DIPROPDWORD** structure.

When the axis mode for the joystick is set to relative, the axis coordinate represents the number of units of movement along the axis since the last value was returned.

Checking for Lost Joystick Input

If you are using the joystick in foreground mode (see Cooperative Levels) you may lose the device when the focus shifts to another application.

You can check for the DIERR_INPUTLOST return value from the **IDirectInputDevice::GetDeviceData** or **IDirectInputDevice::GetDeviceState** methods, and attempt to reacquire the joystick if necessary. (See Acquiring Devices.)

Note You should not attempt to reacquire the joystick on getting a DIERR_NOTACQUIRED error. If you do, you are likely to get caught in an infinite loop: acquisition will fail, you will get another DIERR_NOTACQUIRED error, and so on.

Because access to the joystick is not going to be lost except when your application moves to the background – unlike the mouse and keyboard, the joystick is never used by the Windows system – an alternative to the above method is to reacquire the device in response to a WM_ACTIVATE message.

Force Feedback

Force feedback is the generation of push or resistance in an input/output device, for example by motors mounted in the base of a joystick. DirectInput allows you to generate force feedback effects for devices that have compatible drivers.

The following sections introduce the elements of force feedback:

- Basic Concepts of Force Feedback
- Effect Enumeration
- Information About a Supported Effect
- Creating an Effect
- Effect Direction
- Examples of Setting Effect Direction
- Envelopes and Offsets
- Effect Playback
- Downloading and Unloading Effects
- Changing an Effect
- Gain
- Force Feedback State
- Effect Object Enumeration
- Constant Forces
- Ramp Forces
- Periodic Effects
- Conditions
- Custom Forces
- Device-Specific Effects

To enumerate, create, and manipulate effects, you must first obtain a pointer to the **IDirectInputDevice2** interface for the force feedback device. For an example of how to do this, see Creating the DirectInput Device.

Basic Concepts of Force Feedback

A particular instance of force feedback is called an *effect*, and the push or resistance is called the *force*. Most effects fall into one of the following categories:

- *Constant force*. A steady force in a single direction.
- *Ramp force*. A force that steadily increases or decreases in magnitude.
- *Periodic effect*. A force that pulsates according to a defined wave pattern.
- *Condition*. A force that occurs only in response to input by the user. Two examples are a friction effect that generates resistance to movement of the joystick, and a spring effect that tends to move the stick back to a certain position after it has been moved from that position.

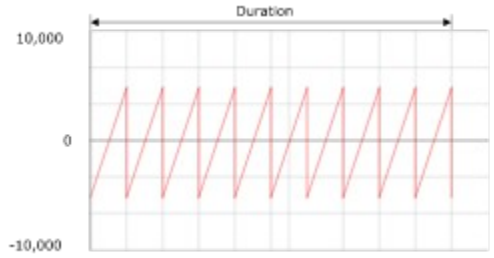
The strength of the force is called its *magnitude*. Magnitude is measured in units ranging from zero (no force) to 10,000 (maximum force for the device, defined in Dinput.h as DI_FFNOMINALMAX). A negative value indicates force in the opposite direction. Magnitudes are linear: a force of 10,000 is twice as great as one of 5,000.

Ramp forces have a beginning and ending magnitude. For a periodic effect, the basic magnitude is the force at the peak of the wave.

The *direction* of a force is the direction from which it comes; just as a north wind comes from the north, a positive force on an given axis pushes from the positive toward the negative.

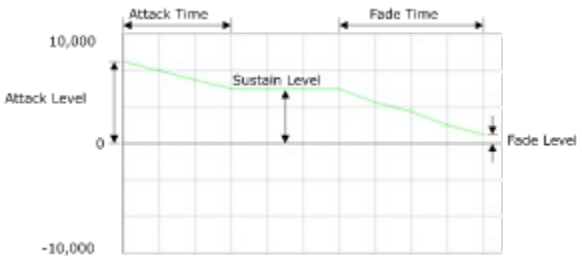
Effects also have *duration*, measured in microseconds. Periodic effects have a *period*, or the duration of one cycle, also measured in microseconds. The *phase* of a periodic effect is the point along the wave where playback begins.

The following diagram represents a sawtooth periodic effect with a magnitude of 5,000, or half the maximum force for the device. The horizontal axis represents the duration of the effect, and the vertical axis represents the magnitude. Points above the center line represent positive force in the direction defined for the effect, and points below the center line represent negative force, or force in the opposite direction.

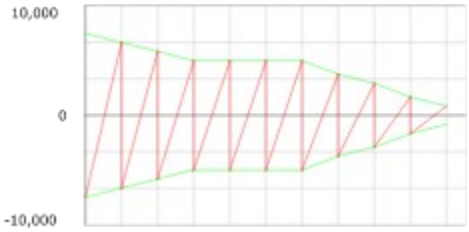


A force may be further shaped by an *envelope*. An envelope defines an *attack value* and a *fade value*, which modify the beginning and ending magnitude of the effect. Attack and fade also have duration, which determines how long the magnitude takes to reach or fall away from the *sustain value*, the magnitude in the middle portion of the effect.

The following diagram represents an envelope. The attack level is set to 8,000 and the fade level to 1,000. The sustain level will be defined by the basic magnitude of the force to which the envelope is being applied; in the example it is 5,000. Note that in this case the attack is greater than the sustain, giving the effect an initial strong kick. Both the attack and the fade level can be either greater or lesser than the sustain level.



The next diagram shows the result of the envelope being applied to the periodic effect in the first diagram. Note that the envelope is mirrored on the negative side of the magnitude. An attack value of 8,000 means that the initial magnitude of the force in either direction will be 80 percent of the maximum possible.



Periodic effects and conditions can also be modified by the addition of an *offset*, which defines the amount by which the waveform is shifted up or down from the base level. The practical effect of applying a positive offset to the sawtooth example would be to strengthen the positive force and weaken the negative one—in other words, the force would peak more strongly in one direction than in the other.

Finally, the overall magnitude of an effect can be scaled by *gain*, which is analogous to a volume control in audio. A single gain value can be applied to all effects for a device; you might want to do this to compensate for stronger or weaker forces on different hardware, or to accommodate the user's preferences.

Effect Enumeration

The **IDirectInputDevice2::EnumEffects** method returns information about the support offered by the device for various kinds of effects.

It is important to distinguish between *supported effects* and *created effects*, or *effect objects*. A supported effect might be a constant force that can be shaped by an envelope. However, this effect has no properties such as magnitude, direction, duration, attack, or fade. You set these properties when you create an effect object in your application. A supported effect may be represented by many effect objects, each with different parameters—for example, several constant forces each with different duration, magnitude, and direction.

For information on enumerating created effects, see Effect Object Enumeration.

Like other DirectInput enumerations, the **IDirectInputDevice2::EnumEffects** method requires a callback function; this is documented with the placeholder name **DIEnumEffectsProc**, but you can use a different name if you wish. This function is called for each effect enumerated. Within the function you can obtain the GUID for each effect, get information about the extent of hardware support, and create one or more effect objects whose methods you can use to manipulate the effect.

Here is a skeletal C++ example of the callback function, and the call to the **IDirectInputDevice2::EnumEffects** method that sets the enumeration in motion. Note that the *pvRef* parameter of the callback can be any 32-bit value; in this case it is a pointer to the device interface, used for getting information about effects supported by the device and for creating effect objects.

```
HRESULT  hr;
// LPDIRECTINPUTDEVICE lpdid2;   // already initialized

BOOL CALLBACK DIEnumEffectsProc(LPCDIEFFECTINFO pdei,
                                LPVOID pvRef)
{
    LPDIRECTINPUTDEVICE2 lpdid = pvRef;   // pointer to calling device
    LPDIRECTINPUTEFFECT  lpdiEffect;      // pointer to created effect
    DIEFFECT             diEffect;        // params for created effect
    DICONSTANTFORCE      diConstantForce; // type-specific parameters
                                          // for diEffect


    if (DIEF_GETTYPE(pdei->dwEffType) == DIEFFT_CONSTANTFORCE)
    {
       /* Here you can extract information about support for the
          effect type (from pdei), and tailor your effects
          accordingly. For example, the device might not support
          envelopes for this type of effect. */
       .
       .
       .
       // Create one or more constant force effects.
       // For each you have to initialize a DICONSTANTFORCE
       // and a DIEFFECT structure.
       // See detailed example at Creating an Effect
       .
       .
       .
        hr = pdid->CreateEffect(pdei->guid,
                                &diEffect,
                                &lpdiEffect,
```

```
                                    NULL);
        .
        .
        .
    }
    // And so on for other types of effect
   .
   .
   .

    return DIENUM_CONTINUE;
}  // end of callback
.
.
.
// Set the callback into motion
hr = lpdid2->EnumEffects(&EnumEffectsProc,
                         lpdid2, DIEFT_ALL);
```

For more information on how to initialize an effect, see Creating an Effect.

Information About a Supported Effect

The **IDirectInputDevice2::GetEffectInfo** method can be used to retrieve information about the device's support for an effect. It retrieves the same information that is returned in the **DIEFFECTINFO** structure during enumeration. For more information, see Effect Enumeration.

The following C++ example fetches information about an enumerated effect whose GUID is stored in the *EffectGuid* variable:

```
DIEFFECTINFO diEffectInfo;
diEffectInfo.dwSize = sizeof(DIEFFECTINFO);
lpdid2->GetEffectInfo(&diEffectInfo, EffectGuid);
```

Creating an Effect

You create an effect object by using the **IDirectInputDevice2::CreateEffect** method, as in the following C++ example, where *pdev2* points to an instance of the interface. This example creates a very simple effect that will pull the joystick away from the user at full force for half a second.

```
HRESULT  hr;
LPDIRECTINPUTEFFECT lpdiEffect;  // receives pointer to created effect
DIEFFECT diEffect;               // parameters for created effect

DWORD    dwAxes[2] = { DIJOFS_X, DIJOFS_Y };
LONG     lDirection[2] = { 18000, 0 };

DICONSTANTFORCE diConstantForce;

diConstantForce.lMagnitude = DI_FFNOMINALMAX;   // full force

diEffect.dwSize = sizeof(DIEFFECT);
diEffect.dwFlags = DIEFF_POLAR | DIEFF_OBJECTOFFSETS;
diEffect.dwDuration = 0.5 * DI_SECONDS;
diEffect.dwSamplePeriod = 0;                // = default
diEffect.dwGain = DI_FFNOMINALMAX;          // no scaling
diEffect.dwTriggerButton = DIEB_NOTRIGGER; // not a button response
diEffect.dwTriggerRepeatInterval = 0;       // not applicable
diEffect.cAxes = 2;
diEffect.rgdwAxes = &dwAxes;
diEffect.rglDirection = &lDirection;
diEffect.lpEnvelope = NULL;
diEffect.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
diEffect.lpvTypeSpecificParams = &diConstantForce;

hr = pdev2->CreateEffect(GUID_ConstantForce,
                         &diEffect,
                         &lpdiEffect,
                         NULL);
```

In the method call, the first parameter identifies the supported effect with which the created effect is to be associated. The example uses one of the predefined GUIDs found in Dinput.h. Note that if you use a predefined GUID, the call will fail if the device doesn't support the effect.

The second parameter sets the parameters as specified in the **DIEFFECT** structure.

The third parameter receives a pointer to the effect object if the call is successful.

The DIEFF_POLAR flag specifies the type of coordinates used for the direction of the force. (See Effect Direction.) It is combined with DIEFF_OBJECTOFFSETS, which indicates that any buttons or axes used in other members will be identified by their offsets within the **DIDATAFORMAT** structure for the device. The alternative is to use the DIEFF_OBJECTIDS flag, signifying that buttons and axes will be identified by the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned for the object when it was enumerated with the **IDirectInputDevice::EnumObjects** method.

For more information on the members of the **DIEFFECT** structure, see Effect Direction.

Effect Direction

Directions can be defined for one or more axes. As with the mouse and joystick, the x-axis increases from left to right, and the y-axis increases from far to near. For three-dimensional devices, the z-axis increases from up to down.

The direction of an effect is the direction from which it comes. An effect with a direction along the negative y-axis tends to push the stick along the positive y-axis (toward the user). It is somewhat easier to visualize the axis values of a direction if you imagine the user exerting a counteracting force on the device. If the user must push the stick toward the left in order to counteract an effect, the effect has a "left" direction; that is, it lies on the negative x-axis.

Direction can be expressed in *polar, spherical,* or *Cartesian* coordinates.

Polar coordinates are expressed as a single angle, in hundredths of degrees clockwise from whatever zero-point, or true north, has been established for the effect. Normally this is the negative y-axis; that is, away from the user. Thus an effect with a polar coordinate of 9,000 normally has a direction of east, or to the user's right, and the user must exert force to the right in order to counteract it.

Spherical coordinates are also in hundredths of degrees but may contain two or more angles; for each angle, the direction is rotated in the positive direction of the next axis. For a three-dimensional device, the first angle would normally be rotated from the positive x-axis toward the positive y-axis (clockwise from east); the second angle would be rotated toward the positive z-axis (down). Thus a force with a direction of (0, 0) would be to the user's right and parallel to the tabletop. A direction of 27,000 for the first angel and 4,500 for the second would be directly away from the user (270 degrees clockwise from east) and angling toward the floor (45 degrees downward from the tabletop); to counteract a force with this direction, the user would have to push forward and down.

Cartesian coordinates are similar to 3-D vectors. If you draw a straight line on graph paper with an origin of (0, 0) at the center of the page, the direction of the line can be defined by the coordinates of any intersection it crosses, regardless of the distance from the origin. A direction of (1, -2) and a direction of (5, -10) are exactly the same.

Note The coordinates used in creating force feedback effects define only direction, not magnitude or distance.

When an effect is created or modified, the **cAxes**, **rgdwAxes**, and **rglDirection** members of the **DIEFFECT** structure are used to specify the direction of the force.

The **cAxes** member simply specifies the number of elements in the arrays pointed to by the next two members.

The array pointed to by **rgdwAxes** identifies the axes. If the DIEFF_OBJECTOFFSETS flag has been set, the axes are identified by the offsets within the data format structure. These offsets are most readily identified by using the DIJOFS_* defines. (For a list of these values, see Joystick Device Constants.)

Finally, the **rglDirection** member specifies the direction of the force.

Note The **cAxes** and **rgdwAxes** members cannot be modified once they have been set. An effect always has the same axis list.

Regardless of whether you are using Cartesian, polar, or spherical coordinates, you must provide exactly as many elements in **rglDirection** as there are axes in the array pointed to by **rgdwAxes**.

In the polar coordinate system, "north" (zero degrees) lies along the vector (0, -1), where the elements of the vector correspond to the elements in the axis list pointed to by **rgdwAxes**. Normally those axes are x and y, so north is directly along the negative y-axis; that is, away from the user. The last element of *lDirection* must be zero.

In the example under Creating an Effect, the direction of a two-dimensional force is defined in polar coordinates. The force has a south direction - it comes from the direction of the user, so that the user has to pull the stick to counteract it. The direction is 180 degrees clockwise from north, and can be assigned as follows:

```
LONG  lDirection[2] = { 18000, 0 };
```

For greater clarity, the assignment could also be expressed this way:

```
LONG  lDirection[2] = { 180 * DI_DEGREES, 0 };
```

For spherical coordinates, presuming that you are working with a three-axis device, the same direction is assigned as follows:

```
LONG  lDirection[3] = { 90 * DI_DEGREES, 0, 0 }
```

The reference for the **DIEFFECT** structure tells us that the first angle is measured in hundredths of degrees from the (1, 0) direction, rotated in the direction of (0, 1); the second angle is measured in hundredths of degrees towards (0, 0, 1). The elements of the vector notation again correspond to elements in the array pointed to by the **rgdwAxes** member. Suppose the elements of this array represent the x, y, and z axes, in that order. The point of origin is at x = 1 and y = 0; that is, to the user's right. The direction of rotation is toward the positive y-axis (0, 1); that is, toward the user, or clockwise. The force in the example is 90 degrees clockwise from the right; that is, south. Because the second element of *lDirection* is 0, there is no rotation on the third axis.

How do you accomplish the same thing with Cartesian coordinates? Presuming you have used the DIEFF_CARTESIAN flag in the **dwFlags** member, you would specify the direction like this:

```
LONG     lDirection[2] = { 0, 1 };
```

Here again the elements of the array correspond to the axes listed in the array pointed to by **rgdwAxes**. The example sets the x-axis to zero and the y-axis to 1; that is, the direction lies directly along the positive y-axis, or to the south.

The theory of effect directions can be difficult to grasp, but the practice is fairly straightforward. For sample code, see Examples of Setting Effect Direction.

Examples of Setting Effect Direction

Single-Axis Effects

Setting up the direction for a single-axis effect is extremely simple, because there is really nothing to specify. You put the DIEFF_CARTESIAN flag in the **dwFlags** member of the **DIEFFECT** structure and set **rglDirection** to point to a single **LONG** containing the value 0.

The following example sets up the direction and axis parameters for an x-axis effect:

```
DIEFFECT eff;
LONG     lZero = 0;                          // No direction
DWORD    dwAxis = DIJOFS_X;                  // x-axis effect

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 1;                               // One axis
eff.dwFlags =
    DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS; // Flags
eff.rglDirection = &lZero;                   // Direction
eff.rgdwAxes = &dwAxis;                      // Axis for effect
```

Two-Axis Effects with Polar Coordinates

Setting up the direction for a polar two-axis effect is only a little more complicated. You set the DIEFF_POLAR flag in **dwFlags** and set **rglDirection** to point to an array of two **LONG**s. The first element in this array is the direction you want the effect to come from. The second element in the array must be zero.

The following example sets up the direction and axis parameters for a two-axis effect coming from the east:

```
DIEFFECT eff;
LONG     rglDirection = { 90 * DI_DEGREES, 0 }; // 90 degrees from
                                                // north, i.e. east
DWORD    rgdwAxes[2] = { DIJOFS_X, DIJOFS_Y };  // x- and y-axis

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 2;                                  // Two axes
eff.dwFlags =
    DIEFF_POLAR | DIEFF_OBJECTOFFSETS;          // Flags
eff.rglDirection = rglDirection;                // Direction
eff.rgdwAxes = rgdwAxes;                        // Axis for effect
```

Two-Axis Effects with Cartesian Coordinates

Setting up the direction for a Cartesian two-axis effect is a bit trickier, but not by much. You set the DIEFF_CARTESIAN flag in **dwFlags** and again set **rglDirection** to point to an array of two **LONG**s. This time the first element in the array is the x-coordinate of the direction vector, and the second is the y-coordinate.

The following example sets up the direction and axis parameters for a two-axis effect coming from the east:

```
DIEFFECT eff;
LONG     rglDirection = { 1, 0 };               // Positive x = east
DWORD    rgdwAxes[2] = { DIJOFS_X, DIJOFS_Y };  // x- and y-axis

ZeroMemory(&eff, sizeof(DIEFFECT));
eff.cAxes = 2;                                  // Two axes
```

```
eff.dwFlags =
    DIEFF_CARTESIAN | DIEFF_OBJECTOFFSETS;      // Flags
eff.rglDirection = rglDirection;                // Direction
eff.rgdwAxes = rgdwAxes;                        // Axis for effect
```

Envelopes and Offsets

You can modify the basic magnitude of some effects by applying an envelope and an offset. For an overview, see Basic Concepts of Force Feedback

To apply an envelope when creating or modifying an effect, initialize a **DIENVELOPE** structure and put a pointer to it in the **lpEnvelope** member of the **DIEFFECT** structure.

The device driver determines which effects support envelopes. Typically you can apply an envelope to a constant force, a ramp force, or a periodic effect, but not to a condition. To determine whether a particular effect supports an envelope, you call the **IDirectInputDevice2::GetEffectInfo** method and check for the DIEP_ENVELOPE flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

To apply an offset, set the **lOffset** member of the **DIPERIODIC** or **DICONDITION** structure pointed to by the **lpvTypeSpecificParams** member of the **DIEFFECT** structure. For periodic effects, the absolute value of the offset plus the magnitude of the effect must not exceed DI_FFNOMINALMAX.

You cannot apply an offset to a constant force or ramp force. In these cases the same effect can be achieved by altering the magnitude.

Effect Playback

There are two principal ways to start playback of an effect: manually by a call to the **IDirectInputEffect::Start** method, and automatically in response to a button press. Playback also starts when you change an effect by calling the **IDirectInputEffect::SetParameters** method with the DIEP_START flag.

Passing INFINITE in the *dwIterations* parameter has the effect of playing the effect repeatedly, with the envelope being applied each time. If you want to repeat an effect without repeating the envelope – for example, to begin with a strong kick, then settle down to a steady throb – set *dwIterations* to 1 and set the **dwDuration** member of the **DIEFFECT** structure to INFINITE. (This is the structure passed to the **IDirectInputDevice2::CreateEffect** method.)

Note Some devices do not support multiple iterations of an effect and accept only the value 1 in the *dwIterations* parameter to the **Start** method. You should always check the return value from **Start** to make sure the effect played successfully.

To associate an effect with a button press, you set the **dwTriggerButton** member of the **DIEFFECT** structure. You also set the **dwTriggerRepeatInterval** member to the desired delay between playbacks when the button is held down; this is the interval, in microseconds, between the end of one playback and the start of the next. On some devices, multiple effects cannot be triggered by the same button; if you associate more than one effect with a button; the last effect downloaded will be the one triggered.

To dissociate an effect from its trigger button, you must either call the **IDirectInputEffect::Unload** method or set the parameters for the effect with **dwTriggerButton** set to DIEB_NOTRIGGER.

Triggered effects, like all others, are lost when the application loses access to the device. In order to make them active again, you must download them as soon as the application reacquires the device. This step is not necessary for effects not associated with a trigger, because they are automatically downloaded if necessary whenever the **Start** method is called.

If an effect has a finite duration and is started by a call to the **Start** method, it will stop playing when the time has elapsed. If its duration was set to INFINITE, playback ends only when the **IDirectInputEffect::Stop** method is called. An effect associated with a trigger button starts when the button is pressed and stops when the button is released or the duration has elapsed, whichever comes sooner.

Downloading and Unloading Effects

Before an effect can be played, it must be downloaded to the device. Downloading an effect means telling the driver to prepare the effect for playback. It is entirely up to the driver to determine how this is done. Generally the driver will place the parameters of the effect in hardware memory in order to minimize the subsequent transfer of data between the device and the system. The consequent reduction in latency is particularly important for conditions and for effects played in response to a trigger, such as a "fire" button. Ideally the device will not have to communicate with the system at all in order to respond to axis movements and button presses.

Downloading is done automatically when you create an effect, provided the device is not full and is acquired at the exclusive cooperative level. By default it is also done when you start the effect or change its parameters. If you specify the DIEP_NODOWNLOAD flag when changing parameters, you must subsequently use the **IDirectInputEffect::Download** method to download or update the effect.

When the device is unacquired – for example, when it has been acquired with the exclusive foreground cooperative level and the application moves to the background – effects are unloaded and must be downloaded again when the application regains the foreground. As stated above, this will be done automatically when you call the **IDirectInputEffect::Start** method, but you may choose to download all effects immediately on reacquiring the device. You always have to download effects associated with a trigger button, since the **Start** method will not normally be called for such effects.

If your application gets the DIERR_DEVICEFULL error when downloading an effect, you have to make room for the new effect by unloading an old one. You can remove an effect from the device by calling the **IDirectInputEffect::Unload** method. You can also remove all effects by resetting the device through a call to **IDirectInputDevice2::SendForceFeedbackCommand**.

When you create a force feedback device, the hardware and driver are reset, so any existing effects are cleared.

Changing an Effect

You can modify the parameters of an effect, in some cases even while the effect is playing. You do this by using the **IDirectInputEffect::SetParameters** method.

The **dwDynamicParams** member of the **DIEFFECTINFO** structure tells you which effect parameters can be changed while an effect is playing. If you attempt to modify an effect parameter that cannot be modified while the effect is playing, and the effect is still playing, then DirectInput will normally stop the effect, update the parameters, and restart the effect. You can override this default behavior by passing the DIEP_NORESTART flag.

The following C++ example changes the magnitude of the constant force that was set in the example under Creating an Effect.

```
DIEFFECT        diEffect;           // parameters for effect
DICONSTANTFORCE diConstantForce;
  // type-specific parameters.

diConstantForce.lMagnitude = 5000;
diEffect.dwSize = sizeof(DIEFFECT);
diEffect.cbTypeSpecificParams = sizeof(DICONSTANTFORCE);
diEffect.lpvTypeSpecificParams = &diConstantForce;
hr = lpdiEffect->SetParameters(&diEffect, DIEP_TYPESPECIFICPARAMS);
```

The flag ensures that the transfer of data from the **DIEFFECT** structure is restricted to the relevant members, so that you do not have to initialize the entire structure and so that the minimum possible amount of data needs to be sent to the device.

Gain

You may want to scale the force of your effects according to the actual force exerted by different devices. For example, if an application's effects feel right on a device that puts out a maximum force of *n* newtons on a given axis, then you may want to adjust the gain for a device that puts out more force. (You cannot use the gain to increase the maximum force of the axis, so you should set the basic effect magnitudes to values suitable for devices that put out less force.)

The actual force generated by a device object such as an axis or button is returned in the **dwFFMaxForce** member of the **DIDEVICEOBJECTINSTANCE** structure when objects are enumerated. (See Device Object Enumeration.)

You can set the gain for the entire device by using the **IDirectInputDevice::SetProperty** method. You can also set the gain for individual effects when creating or modifying them. Put the new gain value in the **dwGain** member of the **DIEFFECT** structure. If modifying the effect with the **SetProperty** method, be sure to include DIEP_GAIN in the flags parameter, unless you are using DIEP_ALLPARAMS, which includes DIEP_GAIN.

The purpose of setting the device gain is to allow your application to have control over the strength of all effects all at once. For example, you might have a slider control in your application to allow the user to specify how strong the force feedback effects should be, like the master volume control on a sound mixer. By setting the device gain, your application won't need to adjust the gain of each individual effect to suit the user's preferences.

A gain value may be in the range 0 to 10,000 (or DI_FFNOMINALMAX), where 10,000 indicates that magnitudes are not to be scaled, 7,500 means that forces are to be scaled to 75 percent of their nominal magnitudes, and so on.

Force Feedback State

The **IDirectInputDevice2::SendForceFeedbackCommand** method allows you to turn off the device's actuators (effectively causing it to ignore any effects that are being played), pause or stop playback of effects, and reset the device so that all downloaded effects are removed.

To retrieve the current force feedback state, use the **IDirectInputDevice2::GetForceFeedbackState** method. This method returns information about whether the actuators are active, whether playback is paused, and whether the device has been reset. It also retrieves information about various switches and about whether the device is currently powered.

Effect Object Enumeration

Whenever you need to examine or manipulate all the effects you have created, you can use the **IDirectInputDevice2::EnumCreatedEffectObjects** method. As no flags are currently defined for this method, you cannot restrict the enumeration to particular kinds of effects; all effects will be enumerated.

Note This method enumerates created effects, not effects supported by a device. For more information on the distinction between the two, see Effect Enumeration.

Like other DirectInput enumerations, the **EnumCreatedEffectObjects** method requires a callback function. This standard callback is documented with the placeholdner name **DIEnumCreatedEffectObjectsProc**, but you can use a different name. The function is called for each effect enumerated. Within the function you can perform any processing you want; however, it is not safe to create a new effect while enumeration is going on.

Here is a skeletal C++ example of the callback function, and the call to the **EnumCreatedEffectObjects** method. Note that the *pvRef* parameter of the callback can be any 32-bit value; in this case it is a pointer to the device interface.

```
HRESULT  hr;
// LPDIRECTINPUTDEVICE lpdid;   // already initialized

BOOL CALLBACK DIEnumCreatedEffectObjectsProc(
                     LPDIRECTINPUTEFFECT peff, LPVOID pvRef);
{
    LPDIRECTINPUTDEVICE  pdid = pvRef; // pointer to calling device
    DIEFFECT            diEffect;      // params for created effect

    diEffect.dwSize = sizeof(DIEFFECTINFO);
    peff->GetParameters(&diEffect, DIEP_ALLPARAMS);
    // check or set parameters, or do anything else
    .
    .
    .
}  // end of callback

// Set the callback into motion
hr = lpdid->EnumCreatedEffectObjects(&EnumCreatedEffectObjectsProc,
                                     &lpdid, 0);
```

Constant Forces

A constant force is a force with a defined magnitude and duration.

You can apply an envelope to a constant force in order to give it shape. For example, suppose you have an effect with a nominal magnitude of 2,000 and a duration of 2 seconds. Then you apply an envelope with the following values:

Attack time	0.5 second
Initial attack level	5,000
Fade time	1 second
Fade level	0

When you play the effect, you get the following:

Elapsed time	**Magnitude**
0.0	5,000
0.1	4,400
0.2	3,800
0.3	3,200
0.4	2,600
0.5	2,000
(duration of sustain)	2,000
1.0:	2,000
1.1	1,800
1.2	1,600
1.3	1,400
1.4	1,200
1.5	1,000
1.6	800
1.7	600
1.8	400
1.9	200
2.0	0

You cannot apply an offset to a constant force.

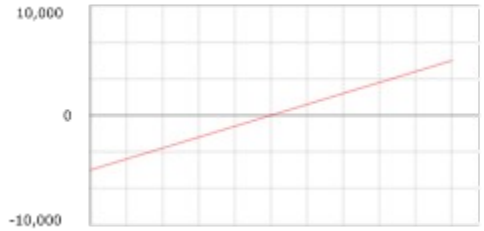
To create a constant force, pass GUID_ConstantForce to the **IDirectInputDevice2::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice2::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT_GETTYPE**(**dwEfftype**)) is equal to DIEFT_CONSTANTFORCE. In this way you can use hardware-specific forces designed by the manufacturer, such as a “constant” force that actually varies in magnitude in a seemingly random fashion to simulate turbulence.

A constant force uses a **DICONSTANTFORCE** structure to define the magnitude of the force, while the duration is taken from the **DIEFFECT** structure. An envelope may be applied.

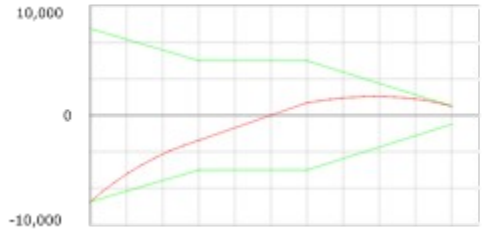
Ramp Forces

A ramp force is a force with defined starting and ending magnitudes and a finite duration. A ramp force may continue in a single direction, or it may start as a strong push in one direction, weaken, stop, and then strengthen in the opposite direction.

The following diagram shows a ramp force that starts at a magnitude of -5,000 and ends at 5,000:



You can apply an envelope to a ramp force in order to shape it further. The following diagram shows the effect of applying an envelope, shown in green, to the ramp force in the previous diagram.



Note that during the sustain portion of the envelope, the magnitude of the effect follows the same straight line as before the envelope was applied. For the duration of the attack and fade, the slope of the ramp is modified by the attack and fade levels.

You cannot apply an offset to a ramp force.

To create a ramp force, pass GUID_RampForce to the **IDirectInputDevice2::CreateEffect** method. You can also pass any other GUID obtained by the **IDirectInputDevice2::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT_GETTYPE**(**dwEfftype**)) is equal to DIEFT_RAMPFORCE. In this way you can use hardware-specific ramp forces designed by the manufacturer.

A ramp force uses a **DIRAMPFORCE** structure to define the starting and ending magnitude of the force, while the duration is taken from the **DIEFFECT** structure. Duration must never be set to INFINITE.

Periodic Effects

Periodic effects are waveform effects. DirectInput defines the following forms:

- *Square*
- *Sine*
- *Cosine*
- *Triangle*
- *SawtoothUp* The waveform drops vertically after it reaches maximum positive force. See the example at Basic Concepts of Force Feedback.
- *SawtoothDown* The waveform rises vertically after it reaches maximum negative force.

An envelope can be applied to periodic effects. See the example at Basic Concepts of Force Feedback.

The phase of a periodic effect is the point along the waveform where the effect begins. Phase is measured in hundredths of a degree, from 0 to 35,999. The following table indicates where selected phase values (in degrees) lie along the various waveforms. *Max* is the top (+) or bottom (-) of the wave and *Mid* is the midpoint, where no force is applied in either direction.

	0	**90**	**180**	**270**
Square	+Max	+Max	-Max	-Max
Sine	Mid	+Max	Mid	-Max
Triangle	+Max	Mid	-Max	Mid
SawtoothUp	-Max	-Max/2	Mid	+Max/2 (reaches +Max just before the cycle repeats)
SawtoothDown	+Max	+Max/2	Mid	-Max/2 (reaches -Max just before the cycle repeats)

A driver may round off a phase value to the nearest supported value. For example, for a sine effect some drivers support only values of 0 and 9,000 (to create a cosine); for other effects, only values of 0 and 18,000 are supported.

To create a periodic force, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice2::CreateEffect** method:

- GUID_Square
- GUID_Sine
- GUID_Triangle
- GUID_SawtoothUp
- GUID_SawtoothDown

You can also pass any other GUID obtained by the **IDirectInputDevice2::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT_GETTYPE**(**dwEfftype**)) is equal to DIEFT_PERIODIC. In this way you can use hardware-specific forces designed by the manufacturer. For example, a hardware device might support a periodic effect that rotates the stick in a small circle.

The type-specific structure for periodic effects is **DIPERIODIC**. For more information on the **dwPhase** member of this structure, see the overview of Periodic Effects.

Conditions

Conditions are forces applied in response to current sensor values within the device. In other words, conditions require information about device motion such as position or velocity of a joystick handle.

In general, conditions are not associated with individual events during a game or other application. They represent ambient phenomena such as the stiffness or looseness of a flight stick, or the tendency of a steering wheel to return to a straight-ahead position.

A condition does not have a predefined magnitude; the magnitude is scaled in proportion to the movement or position of the input object.

DirectInput defines the following types of condition effects:

- *Friction*. The force is applied when the axis is moved, and depends on the defined friction coefficient.
- *Damper*. The force increases in proportion to the velocity with which the user moves the axis.
- *Inertia*. The force increases in proportion to the acceleration of the axis.
- *Spring*. The force increases in proportion to the distance of the axis from a defined neutral point.

Most hardware devices do not support the application of envelopes to conditions. To determine whether a particular effect supports an envelope, check for the DIEP_ENVELOPE flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

Conditions have the following type-specific parameters:

Offset.
The offset from the zero reading of the appropriate sensor value where the condition begins to be applied. For a spring effect, the neutral point—that is, the point along the axis where the spring would be considered at rest—would be defined by the offset for the condition. For a damper, the offset would define the greatest velocity value for which damping force is zero. Offset is not normally used for inertia or friction effects.

Coefficient.
A multiplier that scales the effect. For some devices, you can set separate coefficients for the positive and negative direction along the axis associated with the condition. For example, a flight stick controlling a damaged aircraft might move more easily to the right than to the left.

Saturation.
The saturation point is the point on an axis at which a force is considered to be at its maximum or minimum. Increases or decreases in magnitude beyond this point are ignored. For example, suppose a flight stick has a spring condition on the x-axis. The formula for the condition causes the force to keep increasing with the distance of the axis from the neutral point, but on some devices you could set the positive and negative saturation points so that the full force of the effect is reached when the stick has been moved halfway to the right or left, and does not increase beyond that point.

Deadband.
The deadband is a zone in an axis where a condition does not apply. Taking the spring example again, the deadband would define a zone within which the axis was considered to be at the neutral point, and no force would be applied.

Conditions can have duration, though in most cases you would probably want to set the duration to INFINITE and stop the effect only in response to some event in the application.

To create a condition, pass one of the following values in the *rguid* parameter of the **IDirectInputDevice2::CreateEffect** method:

- GUID_Spring
- GUID_Damper
- GUID_Inertia
- GUID_Friction

You can also pass any other GUID obtained by the **IDirectInputDevice2::EnumEffects** method, provided the low byte of the **dwEffType** member of the **DIEFFECTINFO** structure (**DIEFT_GETTYPE**(**dwEfftype**)) is equal to DIEFT_CONDITION. In this way you can use hardware-specific conditions designed by the manufacturer.

The type-specific structure for conditions is **DICONDITION**. For multiple-axis conditions you may provide an array of such structures, one for each axis, or a single structure that defines the condition in the specified direction. In either case you need to set the **cbTypeSpecificParams** member of the **DIEFFECT** structure to the actual number of bytes used; that is, to **sizeof(DICONDITION)** * *n*, where *n* is the number of structures provided. For more information on how to use either single or multiple structures, see the Remarks for the **DICONDITION** structure.

An application should call **IDirectInputDevice2::GetEffectInfo** or **IDirectInputDevice2::EnumEffects** and examine the **dwEffectType** member of the **DIEFFECTINFO** structure in order to determine if the both a positive and negative coefficient and saturation for the effect are supported on the device. If the effect does not return the DIEFT_POSNEGCOEFFICIENTS flag, it will ignore the value in the **INegativeCoefficient** member and the value in **IPositiveCoefficient** will be applied to the entire axis. Likewise, if the effect does not return the DIEFT_POSNEGSATURATION flag, it will ignore the value in the **dwNegativeSaturation** and the value in **dwPositiveSaturation** will be used as the negative saturation level. Finally, if the effect does not return the DIEFT_SATURATION flag, it will ignore both the **IPositiveSaturation** and **INegativeSaturation** values and no saturation will be applied.

Note that you can set a coefficient to a negative value, and this has the effect of generating the force in the opposite direction. For example, for a spring effect it would cause the spring to push away from the offset point rather than pulling toward it.

You should also check the **dwEffectType** member for the DIEFT_DEADBAND flag, to see if deadband is supported for the condition. If it is not supported, the value in the **IDeadBand** member of the **DICONDITION** structure will be ignored.

Custom Forces

Application writers can create their own effects by creating a custom force. A custom force is an array of constant force values played back by the device.

The type-specific structure for custom waveform effects is **DICUSTOMFORCE**.

You should set the **dwSamplePeriod** member of the **DICUSTOMFORCE** structure and the **dwSamplePeriod** member of the **DIEFFECT** structure to the same value. This is the length of time, in milliseconds, for which each element in the array of forces will be played.

The custom force is played repeatedly until the time set in the **dwDuration** member of the **DIEFFECT** structure has elapsed.

Device-Specific Effects

DirectInput provides a way to control device-specific effects. This is useful for hardware vendors who have extra effects that are not directly supported by DirectInput.

The hardware vendor must provide a GUID identifying the device-specific effect and may provide a custom structure for the type-specific parameters of the effect. Your application then must initialize a **DIEFFECT** structure and a type-specific structure, just as with any other effect. You then call the **IDirectInputDevice2::CreateEffect** method, passing the device-specific GUID and a pointer to the **DIEFFECT** structure.

When you obtain information about a device-specific effect in a **DIEFFECTINFO** structure, the low byte of the **dwEffType** member (**DIEFT_GETTYPE**(**dwEfftype**)) indicates which of the predefined DirectInput effect categories (constant force, ramp force, periodic, or condition) the effect falls into. If it does not fall into any of the predefined categories, then the value is DIEFT_HARDWARE.

If a device-specific effect falls into one of the predefined categories, then the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to the corresponding **DICONSTANTFORCE**, **DIRAMPFORCE**, **DIPERIODIC**, or **DICONDITION** structure, and the **cbTypeSpecificParams** member must be equal to the size of that structure.

If the (**DIEFT_GETTYPE**(**dwEfftype**) == DIEFT_HARDWARE), then the values of the **lpvTypeSpecificParams** and **cbTypeSpecificParams** members depend on whether the effect requires custom type-specific parameters. If it does, then these values must refer to the appropriate structure defined in the manufacturer's header file and declared and initialized by your application. If the effect does not require custom parameters - that is, if the **dwStaticParms** member of the **DIEFFECTINFO** structure for the hardware effect does not have the DIEP_TYPESPECIFICPARAMS flag - then **lpvTypeSpecificParams** must be NULL and **cbTypeSpecificParams** must be zero.

DirectInput passes the GUID and the **LPDIEFFECT** to the device driver for verification. If the GUID is unknown, the device will return DIERR_DEVICENOTREG. If the GUID is known but the type-specific data is incorrect for that effect, the device will return DIERR_INVALIDPARAM.

Designing for Previous Versions of DirectInput

In several places, DirectInput requires you to pass a version number to a method. This parameter specifies which version of DirectInput the DirectInput subsystem should emulate.

Applications designed for the latest version of DirectInput should pass the value DIRECTINPUT_VERSION as defined in Dinput.h.

Applications designed to run under previous versions should pass a value corresponding to the version of DirectInput for which they were designed, with the main version number in the high-order byte. For example, an application that was designed to run on DirectInput 3 should pass a value of 0x0300.

If you define DIRECTINPUT_VERSION as 0x0300 before including the Dinput.h header file, then the header file will generate structure definitions compatible with DirectInput 3.0.

If you do not define DIRECTINPUT_VERSION before including the Dinput.h header file, then the header file will generate structure definitions compatible with the current version of DirectInput. However, the DirectX 3-compatible structures will be available under the same names with "_DX3" appended. For example, the DirectX 3-compatible **DIDEVCAPS** structure is called **DIDEVCAPS_DX3**.

DirectInput Tutorials

This section contains four tutorials, each providing step-by-step instructions for implementing basic DirectInput functionality.

- Tutorial 1: Using the Keyboard

  The first tutorial shows how to add DirectInput keyboard support to an existing application.
- Tutorial 2: Using the Mouse

  The next tutorial takes you through the steps of providing DirectInput mouse support in an application. The tutorial is based on the sample application Scrawl, included in the DirectX code samples under the Platform SDK References, and focuses on buffered data.
- Tutorial 3: Using the Joystick

  This tutorial shows how to enumerate all the joysticks connected to a system, how to create and initialize DirectInputDevice objects for each of them in a callback function, and how to retrieve immediate data. Sample code is from the Space Donuts application, included in the DirectX code samples in the Platform SDK.
- Tutorial 4: Using Force Feedback

  The final tutorial illustrates the creation and manipulation of a simple effect on a force feedback joystick.

  Additional sample code can be found in the Samples folder of the DirectX SDK. The Diex1 through Diex4 programs are simple examples of how to read mouse and keyboard data. The Diff1 program shows how to use force feedback in a very simple way.

Tutorial 1: Using the Keyboard

To prepare for keyboard input, you first create an instance of a DirectInput object. Then you use the **IDirectInput::CreateDevice** method to create an instance of an **IDirectInputDevice** interface. The **IDirectInputDevice** interface methods are used to manipulate the device, set its behavior, and retrieve data.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Creating the DirectInput Object
- Step 2: Creating the DirectInput Keyboard Device
- Step 3: Setting the Keyboard Data Format
- Step 4: Setting the Keyboard Behavior
- Step 5: Gaining Access to the Keyboard
- Step 6: Retrieving Data from the Keyboard
- Step 7: Closing Down the DirectInput System

Adding DirectInput keyboard support to an application is relatively simple, so this tutorial is not accompanied by a complete sample application. All of the tutorial steps are illustrated by code within the text. The related steps for initializing the system are gathered in Sample Function 1: DI_Init. Another function, Sample Function 2: DI_Term**,** is called whenever the system needs to be closed down.

Step 1: Creating the DirectInput Object

The first step in setting up the DirectInput system is to create a single DirectInput object as overall manager. This is done with a call to the **DirectInputCreate** function.

```
// HINSTANCE  hinst; // initialized earlier
HRESULT        hr;
LPDIRECTINPUT  g_lpdi;

hr = DirectInputCreate(hinst, DIRECTINPUT_VERSION, &g_lpdi, NULL);
if FAILED(hr)
{
  // DirectInput not available; take appropriate action
}
```

The first parameter for **DirectInputCreate** is the instance handle to the application or DLL that is creating the object.

The second parameter tells the DirectInput object which version of the DirectInput system should be used. You can design your application to be compatible with earlier versions of DirectInput. For more information, see Designing for Previous Versions of DirectInput.

The third parameter is the address of a variable that will be initialized with a valid **IDirectInput** interface pointer if the call succeeds.

The last parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Most applications will not be using aggregation and so will pass NULL.

Step 2: Creating the DirectInput Keyboard Device

After creating the DirectInput object, your application must create the keyboard object-the device-and retrieve a pointer to an **IDirectInputDevice** interface. The device will perform most of the keyboard-related tasks, using the methods of the interface.

To do this your application must call the **IDirectInput::CreateDevice** method, as shown in Sample Function 1: DI_Init. **CreateDevice** accepts three parameters.

The first parameter is the GUID for the device being created. Since the system keyboard will be used, your application should pass the *GUID_SysKeyboard* value.

The second parameter is the address of a variable that will be initialized with a valid **IDirectInputDevice** interface pointer if the call succeeds.

The third parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Your application will likely not use aggregation, in which case the parameter is NULL.

The following example attempts to retrieve a pointer to an **IDirectInputDevice** interface. If this fails, it calls the **DI_Term** application-defined sample function to deallocate existing DirectInput objects, if any.

Note In all the examples, *g_lpdi* is the initialized pointer to the DirectInput object. The method calls are in the C++ form.

```
HRESULT             hr;
LPDIRECTINPUTDEVICE  g_lpDIDEVICE

hr = g_lpDI->CreateDevice(GUID_SysKeyboard, &g_lpDIDevice, NULL);
if FAILED(hr)
{
    DI_Term();
    return FALSE;
}
```

Step 3: Setting the Keyboard Data Format

After retrieving an **IDirectInputDevice** pointer, your application must set the device's data format, as shown in Sample Function 1: DI_Init. For keyboards, this is a very simple task. Call the **IDirectInputDevice::SetDataFormat** method, specifying the data format provided for your convenience by DirectInput in the *c_dfDIKeyboard* global variable.

The following example attempts to set the data format. If this fails, it calls the **DI_Term** sample function to deallocate existing DirectInput objects, if any.

```
hr = g_lpDIDevice->SetDataFormat(&c_dfDIKeyboard);

if FAILED(hr){
    DI_Term();
    return FALSE;
}
```

Step 4: Setting the Keyboard Behavior

Before your application can gain access to the keyboard, it must set the device's behavior using the **IDirectInputDevice::SetCooperativeLevel** method, as shown in Sample Function 1: DI_Init. This method accepts the handle to the window to be associated with the device. DirectInput does not support exclusive access to keyboard devices, so the DISCL_NONEXCLUSIVE flag must be included in the *dwFlags* parameter.

The following example attempts to set the device's cooperative level. If this fails, it calls the **DI_Term** application-defined sample function to deallocate existing DirectInput objects, if any.

```
// Set the cooperative level
hr = g_lpDIDevice->SetCooperativeLevel(g_hwndMain,
                   DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);

if FAILED(hr){
    DI_Term();
    return FALSE;
}
```

Step 5: Gaining Access to the Keyboard

After your application sets the keyboard's behavior, it can acquire access to the device by calling the **IDirectInputDevice::Acquire** method. The application must acquire the device before retrieving data from it. The **Acquire** method accepts no parameters.

The following line of code acquires the keyboard device that was created in Step 2: Creating the DirectInput Keyboard Device:

```
if (g_lpDIDevice) g_lpDIDevice->Acquire();
```

Step 6: Retrieving Data from the Keyboard

Once a device is acquired, your application can start retrieving data from it. The simplest way to do this is to call the **IDirectInputDevice::GetDeviceState** method, which takes a snapshot of the device's state at the time of the call.

The **GetDeviceState** method accepts two parameters: the size of a buffer to be filled with device state data, and a pointer to that buffer. For keyboards, always declare a buffer of 256 unsigned bytes.

The following sample attempts to retrieve the state of the keyboard. If this fails, it calls an application-defined sample function to deallocate existing DirectInput objects, if any. (See Sample Function 2: DI_Term.)

After retrieving the keyboard's current state, your application may respond to specific keys that were down at the time of the call. Each element in the buffer represents a key. If an element's high bit is on, the key was down at the moment of the call; otherwise, the key was up. To check the state of a given key, use the DirectInput Keyboard Device Constants to index the buffer for a given key.

The following skeleton function, called from the main loop of a hypothetical spaceship game, uses the **IDirectInputDevice::GetDeviceState** method to poll the keyboard. It then checks to see if the LEFT ARROW, RIGHT ARROW, UP ARROW or DOWN ARROW keys were pressed when the device state was retrieved. This is accomplished with the **KEYDOWN** macro defined in the body of the function. The macro accepts a buffer's variable name and an index value, then checks the byte at the specified index to see if the high bit is set and returns TRUE if it is.

```
void WINAPI ProcessKBInput()
{
    #define KEYDOWN(name,key) (name[key] & 0x80)

    char     buffer[256];
    HRESULT  hr;

    hr = g_lpDIDevice->GetDeviceState(sizeof(buffer),(LPVOID)&buffer);
    if FAILED(hr)
    {
         // If it failed, the device has probably been lost.
         // We should check for (hr == DI_INPUTLOST)
         // and attempt to reacquire it here.
         return;
    }

    // Turn the ship right or left
    if (KEYDOWN(buffer, DIK_RIGHT));
        // Turn right.
    else if(KEYDOWN(buffer, DIK_LEFT));
        // Turn left.

    // Thrust or stop the ship
    if (KEYDOWN(buffer, DIK_UP)) ;
        // Move the ship forward.
    else if (KEYDOWN(buffer, DIK_DOWN));
        // Stop the ship.
}
```

Step 7: Closing Down the DirectInput System

When an application is about to close, it should destroy all DirectInput objects. This is a three-step process:

- Unacquire all DirectInput devices (**IDirectInputDevice::Unacquire**)
- Release all DirectInput devices (**IDirectInputDevice::Release**)
- Release the DirectInput object (**IDirectInput::Release**)

For a sample function that closes down the DirectInput system, see Sample Function 2: DI_Term.

Sample Function 1: DI_Init

This application-defined sample function creates a DirectInput object, initializes it, and retrieves the necessary interface pointers, assigning them to global variables. When initialization is complete, it acquires the device.

If any part of the initialization fails, this function calls the **DI_Term** application-defined sample function to deallocate DirectInput objects and interface pointers in preparation for terminating the program. (See Sample Function 2: DI_Term.)

Besides creating the DirectInput object, the **DI_Init** function performs the tasks discussed in the following tutorial steps:

- Step 2: Creating the DirectInput Keyboard Device
- Step 3: Setting the Keyboard Data Format
- Step 4: Setting the Keyboard Behavior
- Step 5: Gaining Access to the Keyboard

Here is the **DI_Init** function:

```
// HINSTANCE            g_hinst;    //initialized application instance
// HWND                 g_hwndMain; //initialized application window
LPDIRECTINPUT        g_lpDI;
LPDIRECTINPUTDEVICE  g_lpDIDevice;

BOOL WINAPI DI_Init()
{
    HRESULT hr;

    // Create the DirectInput object.
    hr = DirectInputCreate(g_hinst, DIRECTINPUT_VERSION,
                           &g_lpDI, NULL);
    if FAILED(hr) return FALSE;

    // Retrieve a pointer to an IDirectInputDevice interface
    hr = g_lpDI->CreateDevice(GUID_SysKeyboard, &g_lpDIDevice, NULL);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

// Now that you have an IDirectInputDevice interface, get
// it ready to use.

    // Set the data format using the predefined keyboard data
    // format provided by the DirectInput object for keyboards.
    hr = g_lpDIDevice->SetDataFormat(&c_dfDIKeyboard);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    // Set the cooperative level
    hr = g_lpDIDevice->SetCooperativeLevel(g_hwndMain,
```

```
                        DISCL_FOREGROUND | DISCL_NONEXCLUSIVE);
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    // Get access to the input device.
    hr = g_lpDIDevice->Acquire();
    if FAILED(hr)
    {
        DI_Term();
        return FALSE;
    }

    return TRUE;
}
```

Sample Function 2: DI_Term

This application-defined sample function deallocates existing DirectInput interface pointers in preparation for program shutdown or in the event of a failure to properly initialize a device.

```
//  LPDIRECTINPUT        g_lpDI;
//  LPDIRECTINPUTDEVICE  g_lpDIDevice;

void WINAPI DI_Term()
{
    if (g_lpDI)
    {
        if (g_lpDIDevice)
        {
            /*
             *  Always unacquire the device before calling Release().
             */
             g_lpDIDevice->Unacquire();
             g_lpDIDevice->Release();
             g_lpDIDevice = NULL;
        }
        g_lpDI->Release();
        g_lpDI = NULL;
    }
}
```

Tutorial 2: Using the Mouse

This tutorial guides you through the process of setting up a mouse device and retrieving buffered input data. The examples are taken from the Scrawl sample application included in the SDK\SAMPLES directory of the DirectX SDK.

To prepare for mouse input, you first create an instance of a DirectInput object. Then you use the **IDirectInput::CreateDevice** method to create an instance of an **IDirectInputDevice** interface. The **IDirectInputDevice** interface methods are used to manipulate the device, set its behavior, and retrieve data.

The preliminary step of setting up the DirectInput system and the final step of closing it down are the same for any application and are covered in Tutorial 1: Using the Keyboard.

This tutorial breaks down the required tasks into the following steps:

- Step 1: Creating the DirectInput Mouse Device
- Step 2: Setting the Mouse Data Format
- Step 3: Setting the Mouse Behavior
- Step 4: Preparing for Buffered Input from the Mouse
- Step 5: Managing Access to the Mouse
- Step 6: Retrieving Buffered Data from the Mouse

Note When an application acquires the mouse at the exclusive cooperative level, Windows does not show a mouse pointer on the screen. For this, your application needs a simple sprite engine. The Scrawl sample application uses the Win32 function **DrawIcon** to display a crosshair cursor.

Step 1: Creating the DirectInput Mouse Device

After creating the DirectInput object, your application should retrieve a pointer to an **IDirectInputDevice** interface, which will be used to perform most mouse-related tasks. To do this, call the **IDirectInput::CreateDevice** method.

The **CreateDevice** method accepts three parameters.

The first parameter is the globally unique identifier (GUID) for the device your application is creating. In this case, since the system mouse will be used, your application should pass the predefined global variable *GUID_SysMouse*.

The second parameter is the address of a variable that will be initialized with a valid **IDirectInputDevice** interface pointer if the call succeeds.

The third parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Your application probably won't be using aggregation, in which case the parameter will be NULL.

The following sample from Scrawl.cpp attempts to retrieve a pointer to an **IDirectInputDevice** interface. If the call fails, an error message is displayed and FALSE is returned.

```
// LPDIRECTINPUT    g_pdi;    // This has been initialized
LPDIRECTINPUTDEVICE g_pMouse;
HRESULT             hr;

hr = g_pdi->CreateDevice(GUID_SysMouse, &g_pMouse, NULL);

if (FAILED(hr)) {
    Complain(hwnd, hr, "CreateDevice(SysMouse)");
    return FALSE;
}
```

Step 2: Setting the Mouse Data Format

After retrieving an **IDirectInputDevice** pointer, your application must set the device's data format. For mouse devices, this is a very simple task. Call the **IDirectInputDevice::SetDataFormat** method, specifying the data format provided for your convenience by DirectInput in the *c_dfDIMouse* global variable.

The following code attempts to set the device's data format. If the call fails, an error message is displayed and FALSE is returned.

```
hr = g_pMouse->SetDataFormat(&c_dfDIMouse);

if (FAILED(hr)) {
    Complain(hwnd, hr, "SetDataFormat(SysMouse, dfDIMouse)");
    return FALSE;
}
```

Step 3: Setting the Mouse Behavior

Before it can gain access to the mouse, your application must set the mouse device's behavior using the **IDirectInputDevice::SetCooperativeLevel** method. This method accepts the handle to the window to be associated with the device. In Scrawl, the DISCL_EXCLUSIVE flag is included to ensure that this application is the only one that can have exclusive access to the device. This flag is combined with DISCL_FOREGROUND because Scrawl is not interested in what the mouse is doing when another application is in the foreground.

The following code from Scrawl.cpp attempts to set the device's cooperative level. If this attempt fails, an error message is displayed and FALSE is returned.

```
hr = g_pMouse->SetCooperativeLevel(hwnd,
                DISCL_EXCLUSIVE | DISCL_FOREGROUND);

if (FAILED(hr)) {
    Complain(hwnd, hr, "SetCooperativeLevel(SysMouse)");
    return FALSE;
}
```

Step 4: Preparing for Buffered Input from the Mouse

The Scrawl application demonstrates how to use event notification to find out about mouse activity, and how to read buffered input from the mouse. Both these techniques require some setup. You can perform these steps at any time after creating the mouse device and before acquiring it.

First, create an event and associate it with the mouse device. You are instructing DirectInput to notify the mouse device object whenever a hardware interrupt indicates that new data is available.

This is how it's done in Scrawl. As usual, the **Complain** function informs the user of any errors.

```
// HANDLE  g_hevtMouse;   // This is global

g_hevtMouse = CreateEvent(0, 0, 0, 0);

if (g_hevtMouse == NULL) {
    Complain(hwnd, GetLastError(), "CreateEvent");
    return FALSE;
}

hr = g_pMouse->SetEventNotification(g_hevtMouse);

if (FAILED(hr)) {
    Complain(hwnd, hr, "SetEventNotification(SysMouse)");
    return FALSE;
}
```

Now you need to set the buffer size so that DirectInput can store any input data until you're ready to look at it. Remember, by default the buffer size is zero, so this step is essential if you want to use buffered data.

It's not necessary to used buffered data with event notification; if you prefer, you can retrieve immediate data when an event is signaled.

To set the buffer size you need to initialize a **DIPROPDWORD** structure with information about itself and about the property you wish to set. Most of the values are boilerplate; the key value is the last one, **dwData**, which is initialized with the number of items you want the buffer to hold.

```
#define DINPUT_BUFFERSIZE  16

DIPROPDWORD dipdw =
    {
        // the header
        {
            sizeof(DIPROPDWORD),        // diph.dwSize
            sizeof(DIPROPHEADER),       // diph.dwHeaderSize
            0,                          // diph.dwObj
            DIPH_DEVICE,                // diph.dwHow
        },
        // the data
        DINPUT_BUFFERSIZE,              // dwData
    };
```

You then pass the address of the header (the **DIPROPHEADER** structure within the **DIPROPDWORD** structure), along with the identifier of the property you want to change, to the **IDirectInputDevice::SetProperty** method, as follows:

```
hr = g_pMouse->SetProperty(DIPROP_BUFFERSIZE, &dipdw.diph);
```

```
if (FAILED(hr)) {
    Complain(hwnd, hr, "Set buffer size(SysMouse)");
    return FALSE;
 }
```

The setup is now complete, and you're ready to acquire the mouse and start collecting data.

Step 5: Managing Access to the Mouse

DirectInput provides the **IDirectInputDevice::Acquire** and **IDirectInputDevice::Unacquire** methods to manage device access. Your application must call the **Acquire** method to gain access to the device before requesting mouse information with the **IDirectInputDevice::GetDeviceState** and **IDirectInputDevice::GetDeviceData** methods.

Most of the time your application will have the device acquired. However, if you have only foreground access the mouse will automatically be unacquired whenever your application moves to the background. You are responsible for reacquiring it when you get the focus back again. This can be done in response to a WM_ACTIVATE message.

Scrawl handles this message by setting a global variable, *g_fActive*, according to whether the application is gaining or losing the focus. It then calls a helper function, **Scrawl_SyncAcquire**, which acquires the mouse if *g_fActive* is TRUE and unacquires it otherwise.

```
case WM_ACTIVATE:
        g_fActive = wParam == WA_ACTIVE || wParam == WA_CLICKACTIVE;
        Scrawl_SyncAcquire(hwnd);
        break;
```

If you have exclusive access, your application may need to let go of the mouse to let the user interact with Windows - for example, to access a menu or a dialog box. In Scrawl this can happen when the user opens the system menu with ALT+SPACEBAR.

The Scrawl window procedure has a handler for WM_ENTERMENULOOP that responds by setting the global variable *g_fActive* to FALSE and calling the **Scrawl_SyncAcquire** function. This handler allows Windows to have the mouse and display its own cursor.

When the user is done using a menu, Windows sends the application a WM_EXITMENULOOP message. In this case, the Scrawl window process posts an application-defined message, WM_SYNCACQUIRE, to its own message queue. This allows other pending messages to be processed before the mouse is reacquired with the **Scrawl_SyncAcquire** function.

Scrawl also unacquires the mouse in response to a right button click, which opens up a context menu. Although the mouse would get unacquired later, in the WM_ENTERMENULOOP handler, it's done here first so that the position of the Windows cursor can be set before the menu appears.

Finally, Scrawl tries to reacquire the mouse if it receives a DIERR_INPUTLOST error after an attempt to retrieve data. This is just in case the device has been unacquired by some mechanism not covered elsewhere; for instance, if the user has pressed CTRL+ALT+DEL.

In summary, your application needs to acquire the mouse before it can get data from it. This needs to be done only once, as long as nothing happens to force your application to give up access to it. In exclusive mode, you are responsible for giving up control of the mouse when Windows needs it. You are also responsible for reacquiring the mouse whenever your program needs access to it after losing it to Windows or another application.

Step 6: Retrieving Buffered Data from the Mouse

Once the mouse is acquired, your application can begin to retrieve data from it.

In the Scrawl sample, retrieval is triggered by a signaled event. In the **WinMain** function, the application sleeps until **MsgWaitForMultipleObjects** indicates that there is either a signal or a message. If there's a signal associated with the mouse, the **Scrawl_OnMouseInput** function is called. This function is a good illustration of how buffered input is handled, so we'll look at it in detail.

First the function makes sure the old cursor position will be cleaned up. Remember, Scrawl is maintaining its own cursor and is wholly responsible for drawing and erasing it.

```
void Scrawl_OnMouseInput(HWND hwnd)
{
    /* Invalidate the old cursor so it will be erased */
    InvalidateCursorRect(hwnd);
```

Now the function enters a loop to read and respond to the entire contents of the buffer. Because it retrieves just one item at a time, it needs only a single **DIDEVICEOBJECTDATA** structure to hold the data.

Another way to go about handling input would be to read the entire buffer at once and then loop through the retrieved items, responding to each one in turn. In that case, *dwElements* would be the size of the buffer, and *od* would be an array with the same number of elements.

```
    while (!fDone) {
        DIDEVICEOBJECTDATA od;
        DWORD dwElements = 1;   // number of items to be retrieved
```

The application calls the **IDirectInputDevice::GetDeviceData** method in order to fetch the data. The second parameter tells DirectInput where to put the data, and the third tells it how many items are wanted. The final parameter would be DIGDD_PEEK if the data was to be left in the buffer, but in this case the data is not going to be needed again, so it is removed.

```
        HRESULT hr = g_pMouse->GetDeviceData(
                            sizeof(DIDEVICEOBJECTDATA),
                            &od,
                            &dwElements, 0);
```

Now the application checks to see if access to the device has been lost and, if so, tells itself to try to reacquire the mouse at the first opportunity. This step was discussed in Step 5: Managing Access to the Mouse.

```
        if (hr == DIERR_INPUTLOST) {
            PostMessage(hwnd, WM_SYNCACQUIRE, 0, 0L);
            break;
        }
```

Next the application makes sure the call to the **GetDeviceData** method succeeded and that there was actually data to be retrieved. Remember, after the call to **GetDeviceData** the *dwElements* variable shows how many items were actually retrieved.

```
        /* Unable to read data or no data available */
        if (FAILED(hr) || dwElements == 0) {
            break;
        }
```

If execution has proceeded to this point, everything is fine: the call succeeded and there is an item of data in the buffer. Now the application looks at the **dwOfs** member of the **DIDEVICEOBJECTDATA**

structure to determine which object on the device reported a change of state, and calls helper functions to respond appropriately. The value of the **dwData** member, which gives information about what happened, is passed to these functions.

```
        /* Look at the element to see what happened */

        switch (od.dwOfs) {

        /* DIMOFS_X: Mouse horizontal motion */
        case DIMOFS_X: UpdateCursorPosition(od.dwData, 0); break;

        /* DIMOFS_Y: Mouse vertical motion */
        case DIMOFS_Y: UpdateCursorPosition(0, od.dwData); break;

        /* DIMOFS_BUTTON0: Button 0 pressed or released */
        case DIMOFS_BUTTON0:

            if (od.dwData & 0x80) { /* Button pressed */
                fDone = 1;
                Scrawl_OnButton0Down(hwnd); /* Go into button-down
                                               mode */
            }
            break;

        /* DIMOFS_BUTTON1: Button 1 pressed or released */
        case DIMOFS_BUTTON1:

            if (!(od.dwData & 0x80)) {    /* Button released */
                fDone = 1;
                Scrawl_OnButton1Up(hwnd); /* Context menu time */
            }
        }

    }
```

Finally, the **Scrawl_OnMouseInput** function invalidates the screen rectangle occupied by the cursor, in case the cursor has been moved by one of the helper functions.

```
    /* Invalidate the new cursor so it will be drawn */
    InvalidateCursorRect(hwnd);
}
```

Scrawl also collects mouse data in the **Scrawl_OnButton0Down** function. This is where the application keeps track of mouse movements while the primary button is being held down – that is, while the user is drawing. This function does not rely on event notification, but repeatedly polls the DirectInput buffer until the button is released.

A key point to note in the **Scrawl_OnButton0Down** function is that no actual drawing is done until all pending data has been read. The reason is that each horizontal or vertical movement of the mouse is reported as a separate event. (Both events are, however, placed in the buffer at the same time.) If a line were immediately drawn in response to each separate axis movement, a diagonal movement of the mouse would produce two lines at right angles.

Another way you can be sure that the movement in both axes is taken into account before responding in your application is to check the sequence numbers of the x-axis item and the y-axis item. If the numbers are the same, the two events took place simultaneously. For more information, see Time Stamps and Sequence Numbers.

Tutorial 3: Using the Joystick

This tutorial shows you how to enumerate joysticks on a system and set up two or more joysticks for input. Code samples are based on the Space Donuts application in the SDK\SAMPLES directory of the DirectX SDK. The method calls are in the C form.

The preliminary step of setting up the DirectInput system and the final step of closing it down are the same for any application and are covered in Tutorial 1: Using the Keyboard.

The first step in the tutorial is to enumerate devices; that is, to see what joysticks are available. As part of this process you initialize each joystick device and set its desired characteristics. You then use the **IDirectInputDevice** interface methods to retrieve data from each joystick.

The tutorial breaks down the required tasks into the following steps:

- Step 1: Enumerating the Joysticks
- Step 2: Creating the DirectInput Joystick Device
- Step 3: Setting the Joystick Data Format
- Step 4: Setting the Joystick Behavior
- Step 5: Gaining Access to the Joystick
- Step 6: Retrieving Data from the Joystick

Step 1: Enumerating the Joysticks

After creating the DirectInput system, call the **IDirectInput::EnumDevices** method to enumerate the joysticks. The following code from Input.c in the Space Donuts source directory accomplishes this.

```
//  LPDIRECTINPUT  pdi;  // previously initialized

pdi->lpVtbl->EnumDevices(pdi, DIDEVTYPE_JOYSTICK,
                         InitJoystickInput, pdi, DIEDFL_ATTACHEDONLY);
```

The method call is in the C form. Note that you could use the **IDirectInput_EnumDevices** macro to simplify the call. All DirectInput methods have corresponding macros defined in Dinput.h that expand to the appropriate C or C++ syntax.

The DIDEVTYPE_JOYSTICK constant, passed as the second parameter, specifies the type of device to be enumerated.

InitJoystickInput is the address of a callback function to be called each time a joystick is found; this is where the individual devices will be initialized in the following steps of the tutorial.

The fourth parameter can be any 32-bit value that you want to make available to the callback function. In this case it's a pointer to the DirectInput interface, which the callback function needs to know so it can call **IDirectInput::CreateDevice**.

The last parameter, DIEDFL_ATTACHEDONLY, is a flag that restricts enumeration to devices that are attached to the computer.

Step 2: Creating the DirectInput Joystick Device

After creating the DirectInput object, the application must retrieve a pointer to an **IDirectInputDevice** interface, which will be used to perform most joystick-related tasks. In the Space Donuts sample, this is done in the callback function **InitJoystickInput**, which is called each time a joystick is enumerated.

Here is the first part of the callback function:

```
BOOL FAR PASCAL InitJoystickInput(LPCDIDEVICEINSTANCE pdinst,
                                  LPVOID pvRef)
{
   LPDIRECTINPUT pdi = pvRef;
   LPDIRECTINPUTDEVICE pdev;

   // create the DirectInput joystick device
   if (pdi->lpVtbl->CreateDevice(pdi, &pdinst->guidInstance,
                                 &pdev, NULL) != DI_OK)
   {
      OutputDebugString("IDirectInput::CreateDevice FAILED\n");
      return DIENUM_CONTINUE;
   }
```

The parameters to **InitJoystickInput** are:

- A pointer to the device instance, supplied by the DirectInput system when the device is enumerated.
- A pointer to the DirectInput interface, which you supplied as an parameter to **IDirectInput::EnumDevices**. This parameter could have been any 32-bit value but in this case you want the DirectInput interface so that you can call the **IDirectInput::CreateDevice** method.

The **InitJoystickInput** function declares a local pointer to the DirectInput object, *pdi*, and assigns it the value passed into the callback. It also declares a local pointer to a DirectInput device, *pdev*, which is initialized when the device is created. This device starts life as an instance of the **IDirectInputDevice** interface, but when it is added to the application's list of input devices it is converted to an **IDirectInputDevice2** object so that it can use the **IDirectInputDevice2::Poll** method.

The first task of the callback function, then, is to create the device. The **IDirectInput::CreateDevice** method accepts four parameters.

The first, unnecessary in C++, is a this pointer to the calling DirectInput interface.

The second parameter is a reference to the globally unique identifier (GUID) for the instance of the device. In this case, the GUID is taken from the **DIDEVICEINSTANCE** structure supplied by DirectInput when it enumerated the device.

The third parameter is the address of the variable that will be initialized with a valid **IDirectInputDevice** interface pointer if the call succeeds.

The fourth parameter specifies the address of the controlling object's **IUnknown** interface for use in COM aggregation. Space Donuts doesn't use aggregation, so the parameter is NULL.

Note that if for some reason the device interface cannot be created, DIENUM_CONTINUE is returned from the callback function. This flag instructs DirectInput to keep enumerating as long as there are devices to be enumerated.

Step 3: Setting the Joystick Data Format

Now that the application has a pointer to a DirectInput device, it can call the **IDirectInputDevice** methods to manipulate that device. The first step, which is an essential one, is to set the data format for the joystick. This step tells DirectInput how to format the input data.

The Space Donuts sample performs this action inside the callback function introduced in the previous step.

```
if (pdev->lpVtbl->SetDataFormat(pdev, &c_dfDIJoystick) != DI_OK)
{
   OutputDebugString("IDirectInputDevice::SetDataFormat FAILED\n");
   pdev->lpVtbl->Release(pdev);
   return DIENUM_CONTINUE;
}
```

The *pdev* variable is a pointer to the device interface created by **IDirectInput::CreateDevice**.

The **IDirectInputDevice::SetDataFormat** method takes two parameters. The first is a this pointer to the calling instance of the interface and is unnecessary in C++. The second is a pointer to a **DIDATAFORMAT** structure containing information about how the data for the device is to be formatted. For the joystick, the predefined global variable *c_dfDIJoystick* can be used here.

As in the previous step, the callback function returns DIENUM_CONTINUE if it fails to initialize the device. This flag instructs DirectInput to keep enumerating as long as there are devices to be enumerated.

Step 4: Setting the Joystick Behavior

The joystick device has been created, and its data format has been set. The next step is to set its cooperative level. In the Space Donuts sample, this is done in the callback function called when the device is enumerated. As in the previous step, *pdev* is a pointer to the device interface.

```
if(pdev->lpVtbl->SetCooperativeLevel(pdev, hWndMain,
   DISCL_NONEXCLUSIVE | DISCL_FOREGROUND) != DI_OK)
{
   OutputDebugString("IDirectInputDevice::SetCooperativeLevel
                     FAILED\n");
   pdev->lpVtbl->Release(pdev);
   return DIENUM_CONTINUE;
}
```

Once again, the first parameter to **IDirectInputDevice::SetCooperativeLevel** is a this pointer.

The second parameter is a window handle. In this case the handle to the main program window is passed in.

The final parameter is a combination of flags describing the desired cooperative level. Space Donuts requires input from the joystick only when it is the foreground application, and does not care whether another program is using the joystick in exclusive mode, so the flags are set to DISCL_NONEXCLUSIVE | DISCL_FOREGROUND. (See Cooperative Levels for a full explanation of these flags.)

The final step carried out for each joystick enumerated in the callback function is to set the properties of the device. In the sample, the properties changed include the range and the dead zone for both the x-axis and y-axis.

By setting the range, you are telling DirectInput what maximum and minimum values you want returned for an axis. If you set a range of -1,000 to +1,000 for the x-axis, as in the example, you are asking that a value of -1,000 be returned when the stick is at the extreme left, +1,000 when it is at the extreme right, and zero when it is in the middle.

The dead zone is a region of tolerance in the middle of the axis, measured in ten-thousandths of the physical range of axis travel. If you set a dead zone of 1,000 for the x-axis, you are saying that the stick can travel one-tenth of its range to the left or right of center before a non-center value will be returned. For more information on the dead zone, see Interpreting Joystick Axis Data.

Here's the code to set the range of the x-axis:

```
DIPROPRANGE diprg;

diprg.diph.dwSize       = sizeof(diprg);
diprg.diph.dwHeaderSize = sizeof(diprg.diph);
diprg.diph.dwObj        = DIJOFS_X;
diprg.diph.dwHow        = DIPH_BYOFFSET;
diprg.lMin              = -1000;
diprg.lMax              = +1000;

if(pdev->lpVtbl->SetProperty(pdev, DIPROP_RANGE, &diprg.diph)
                            != DI_OK)
{
   OutputDebugString("IDirectInputDevice::SetProperty(DIPH_RANGE)
                     FAILED\n");
   pdev->lpVtbl->Release(pdev);
   return FALSE;
}
```

The first task here is to set up the **DIPROPRANGE** structure *diprg*, whose address will be passed into the **IDirectInputDevice::SetProperty** method. Actually, it's not the address of the structure itself that is passed but rather the address of its first member, which is a **DIPROPHEADER** structure. See Device Properties for more information.

The property header is initialized with the following values:

- The size of the property structure
- The size of the header structure
- The value returned by the **DIJOFS_X** macro, which points to the object whose property is being changed
- A flag to indicate how the third parameter is to be interpreted

The **lmin** and **lmax** members of the **DIPROPRANGE** structure are assigned the desired range values.

The application now calls the **IDirectInputDevice::SetProperty** method. As usual, the first parameter is a this pointer. The second parameter is a flag indicating which property is being changed. The third parameter is the address of the **DIPROPHEADER** member of the property structure.

Setting the dead zone of the x-axis requires a similar procedure. The Space Donuts sample uses a helper function, **SetDIDwordProperty**, to initialize a **DIPROPDWORD** property structure. Unlike **DIPROPRANGE**, this structure contains only one data member, which in the example is set to 5,000, indicating that the stick must move half of its range from the center before the axis is reported to be off-center.

```
// set X axis dead zone to 50% (to avoid accidental turning)
if (SetDIDwordProperty(pdev, DIPROP_DEADZONE, DIJOFS_X,
                       DIPH_BYOFFSET, 5000) != DI_OK)
{
   OutputDebugString("IDirectInputDevice::
                      SetProperty(DIPH_DEADZONE) FAILED\n");
   pdev->lpVtbl->Release(pdev);
   return FALSE;
}
```

Step 5: Gaining Access to the Joystick

After your application sets a joystick's behavior, it can acquire access to the device by calling the **IDirectInputDevice::Acquire** method. The application must acquire the device before retrieving data from it. The **Acquire** method accepts no parameters.

The Space Donuts application takes care of acquisition in the **ReacquireInput** function. This function does double duty, serving both to acquire the device on startup and to reacquire it if for some reason a DIERR_INPUTLOST error is returned when the application tries to get input data.

In the following code, *g_pdevCurrent* is a global pointer to whatever DirectInput device is currently in use.

```
BOOL ReacquireInput(void)
{
    HRESULT hRes;

    // if we have a current device
    if (g_pdevCurrent)
    {
       // acquire the device
       hRes = IDirectInputDevice_Acquire(g_pdevCurrent);
       // The call above is a macro that expands to:
       // g_pdevCurrent->lpVtbl->Acquire(g_pdevCurrent);

       if (SUCCEEDED(hRes))
       {
          // acquisition successful
          return TRUE;
       }
       else
       {
          // acquisition failed
          return FALSE;
       }
    }
    else
    {
       // we don't have a current device
       return FALSE;
    }
}
```

In this example, acquisition is effected by a call to **IDirectInputDevice_Acquire**, a macro defined in Dinput.h that simplifies the C call to the **IDirectInputDevice::Acquire** method.

Step 6: Retrieving Data from the Joystick

Since your application is more likely concerned with the position of the joystick axes than with their movement, you will probably want to retrieve immediate rather than buffered data from the device. You can do this by polling with **IDirectInputDevice::GetDeviceState**. Remember, not all device drivers will notify DirectInput when the state of the device changes, so it's always good policy to call the **IDirectInputDevice2::Poll** method before checking the device state.

The Space Donuts application calls the following function on each pass through the rendering loop, provided the joystick is the active input device.

```
DWORD ReadJoystickInput(void)
{
    DWORD                   dwKeyState;
    HRESULT                 hRes;
    DIJOYSTATE              js;

    // poll the joystick to read the current state
    hRes = IDirectInputDevice2_Poll(g_pdevCurrent);

    // get data from the joystick
    hRes = IDirectInputDevice_GetDeviceState(g_pdevCurrent,
                                             sizeof(DIJOYSTATE), &js);

    if (hRes != DI_OK)
    {
      // did the read fail because we lost input for some reason?
      // if so, then attempt to reacquire. If the second acquire
      // fails, then the error from GetDeviceData will be
      // DIERR_NOTACQUIRED, so we won't get stuck an infinite loop.
      if(hRes == DIERR_INPUTLOST)
         ReacquireInput();

      // return the fact that we did not read any data
      return 0;
    }

    // Now study the position of the stick and the buttons.

    dwKeyState = 0;
    if (js.lX < 0) {
      dwKeyState |= KEY_LEFT;
    } else if (js.lX > 0) {
      dwKeyState |= KEY_RIGHT;
    }

    if (js.lY < 0) {
      dwKeyState |= KEY_UP;
    } else if (js.lY > 0) {
       dwKeyState |= KEY_DOWN;
   }

    if (js.rgbButtons[0] & 0x80) {
      dwKeyState |= KEY_FIRE;
    }
```

```
    if (js.rgbButtons[1] & 0x80) {
      dwKeyState |= KEY_SHIELD;
    }

    if (js.rgbButtons[2] & 0x80) {
      dwKeyState |= KEY_STOP;
    }

    return dwKeyState;
}
```

Note the calls to **IDirectInputDevice2_Poll** and **IDirectInputDevice_GetDeviceState**. These are macros that expand to C calls to the corresponding methods, similar to the macro in the previous step of this tutorial. The parameters to the macro are the same as those you would pass to the method. Here is what the call to **GetDeviceState** looks like:

```
   hRes = IDirectInputDevice_GetDeviceState(g_pdevCurrent,
                                            sizeof(DIJOYSTATE), &js);
```

The first parameter is the this pointer; that is, a pointer to the calling object. The second parameter is the size of the structure in which the data will be returned, and the last parameter is the address of this structure, which is of type **DIJOYSTATE**. This structure holds data for up to six axes, 32 buttons, and a point-of-view hat. The sample program looks at the state of two axes and three buttons.

If the position of an axis is reported as non-zero, that axis is outside the dead zone, and the function responds by setting the *dwKeyState* variable appropriately. This variable holds the current set of user commands as entered with either the keyboard or the joystick. For example, if the x-axis of the stick is greater than zero, that is considered the same as the RIGHT ARROW key being down.

Joystick buttons work just like keys or mouse buttons: if the high bit of the returned byte is set, the button is down.

Tutorial 4: Using Force Feedback

This tutorial takes you through the process of creating, playing, and modifying a simple effect on a force feedback joystick. The effect is something like a balky chain saw that you're trying to get started. The sample code uses C++ syntax.

The preliminary step of setting up the DirectInput system and the final step of closing it down are essentially the same for any application and are covered in Tutorial 1: Using the Keyboard. However, when closing down the DirectInput force feedback system you must take the additional step of releasing any effects you have created.

The tutorial breaks down the required tasks into the following steps:

Step 1: Enumerating Force Feedback Devices

The first step is to ensure that a force feedback device is available on the system. You do this by calling the **IDirectInput::EnumDevices** method. In the following example, the global pointer to the game device interface is initialized only if the enumeration has succeeded in finding at least one suitable device:

```
LPDIRECTINPUTDEVICE2  g_lpdid2Game = NULL;

lpdi->EnumDevices(DIDEVTYPE_JOYSTICK,
                  DIEnumDevicesProc,
                  NULL,
                  DIEDFL_FORCEFEEDBACK | DIEDFL_ATTACHEDONLY);
if (g_lpdid2Game == NULL)
  {
  // no force feedback joystick available; take appropriate action
  }
```

In the example, *lpdi* is an initialized pointer to the **IDirectInput** interface. The first parameter to **EnumDevices** restricts the enumeration to joystick-type devices. The second parameter is the callback function that's going to be called each time DirectInput identifies a device that qualifies for enumeration. The third parameter is for user-defined data to be passed in or out of the callback function; in this case it's not used. Finally, the flags restrict the enumeration further to devices actually attached to the system that support force feedback.

The callback function is a convenient place to initialize the device as soon as it has been found. (It's assumed that the first device found is the one you want to use.) You'll do this in Step 2: Creating the DirectInput Force Feedback Device.

Step 2: Creating the DirectInput Force Feedback Device

In order to have DirectInput enumerate devices, you must create a callback function of the same type as **DIEnumDevicesProc**. In Step 1 you passed the address of this function to the **IDirectInput::EnumDevices** method.

DirectInput passes into the callback, as the first parameter, a pointer to a **DIDEVICEINSTANCE** structure that tells you what you need to know about the device. The structure member of chief interest in the example is **guidInstance**, the unique identifier for the particular piece of hardware on the user's system You will need to pass this GUID to the **IDirectInput::CreateDevice** method.

Here's the first part of the callback, which extracts the GUID and creates the device object:

```
BOOL CALLBACK DIEnumDevicesProc(LPCDIDEVICEINSTANCE lpddi,
                                LPVOID pvRef)
  {
  HRESULT hr1, hr2;
  LPDIRECTINPUTDEVICE lpdidGame;
  GUID DeviceGuid = lpddi->guidInstance;

  // create game device

  hr1 = lpdi->CreateDevice(DeviceGuid, &lpdidGame, NULL);
```

Note that the pointer to the **IDirectInputDevice** object, *lpdidGame*, is a local variable. You're not going to keep it, because in order to create force feedback effects you need to obtain a pointer to the **IDirectInputDevice2** interface, as follows:

```
  if (SUCCEEDED(hr1))
    {
    hr2 = lpdidGame->QueryInterface(IID_IDirectInputDevice2,
                                    (void **) &g_lpdid2Game);
    lpdidGame->Release();
    }
  else
    {
    OutputDebugString("Failed to create device.\n");
    return DIENUM_STOP;
    }
  if (FAILED(hr2))
    {
    OutputDebugString("Failed to obtain interface.\n");
    return DIENUM_STOP;
    }
```

The next steps, still within the callback function, are similar to those for setting up any input device. Note that you need the exclusive cooperative level for any force feedback device. Since the joystick will be used for input as well as force feedback, you also need to set the data format.

```
 // set cooperative level
  if (FAILED(g_lpdid2Game->SetCooperativeLevel(hMainWindow,
          DISCL_EXCLUSIVE | DISCL_FOREGROUND)))
    {
    OutputDebugString(
      "Failed to set cooperative level.\n");
    lpdid2Game->Release();
    lpdi2Game = NULL;
```

```
    return DIENUM_STOP;
    }

  // set game data format
  if (FAILED(g_lpdid2Game->SetDataFormat(&c_dfDIJoystick)))
    {
    OutputDebugString("Failed to set game device data format.\n");
    lpdid2Game->Release();
    lpdid2Game = NULL;
    return DIENUM_STOP;
    }
```

Finally. you may want to turn off the device's autocenter feature. Autocenter is essentially a condition effect that uses the motors to simulate the springs in a standard joystick. Turning it off gives you more control over the device.

```
  DIPROPDWORD DIPropAutoCenter;

  DIPropAutoCenter.diph.dwSize = sizeof(DIPropAutoCenter);
  DIPropAutoCenter.diph.dwHeaderSize = sizeof(DIPROPHEADER);
  DIPropAutoCenter.diph.dwObj = 0;
  DIPropAutoCenter.diph.dwHow = DIPH_DEVICE;
  DIPropAutoCenter.dwData = 0;

  if (FAILED(lpdid2Game->SetProperty(DIPROP_AUTOCENTER,
                                     &DIPropAutoCenter.diph)))
    {
    OutputDebugString("Failed to change device property.\n");
    }

  return DIENUM_STOP;   // One is enough.
  } // end DIEnumDevicesProc
```

Before using the device, you must acquire it. See Step 5: Gaining Access to the Joystick in the previous tutorial for an example of how to handle acquisition.

Step 3: Enumerating Supported Effects

Now that you've successfully enumerated and created a force feedback device, you can enumerate the effect types it supports.

Effect enumeration is not strictly necessary if you want to create only standard effects that will be available on any device, such as constant forces. When creating the effect object, you can identify the desired effect type simply by using one of the predefined GUIDs, such as GUID_ConstantForce. (For a complete list of these identifiers, see **IDirectInputDevice2::CreateEffect**.)

Another, more flexible approach is to enumerate supported effects of a particular type, and obtain the GUID for the effect from the callback function. This is the approach taken in the FFDonuts sample application in the DirectX code samples in the Platform SDK, and you'll adopt it here as well. You could, of course, use the callback to obtain more information about the device's support for the effect - for example, whether it supports an envelope - but in this tutorial you'll get only the effect GUID.

First, create the callback function that will be called by DirectInput for each effect enumerated. For information on this standard callback, see **DIEnumEffectsProc**. You can give the function any name you like.

```
BOOL  EffectFound = FALSE;  // global flag

BOOL CALLBACK DIEnumEffectsProc(LPCDIEFFECTINFO pei, LPVOID pv)
  {
  *((GUID *)pv) = pei->guid;
  EffectFound = TRUE;
  return DIENUM_STOP;  // one is enough
  }
```

The **GUID** variable pointed to by the application-defined value *pv* is assigned the value passed in the **DIEFFECTINFO** structure created by DirectInput for the effect.

In order to obtain the effect GUID, you set the callback in motion by calling the **IDirectInputDevice2::EnumEffects** method, as follows:

```
HRESULT  hr;
GUID     guidEffect;

hr = g_lpdid2Game->EnumEffects(
     (LPDIENUMEFFECTSCALLBACK) DIEnumEffectsProc,
     &guidEffect,
     DIEFT_PERIODIC);
if (FAILED(hr))
  {
  OutputDebugString("Effect enumeration failed\n");
  // Note: success doesn't mean any effects were found,
  // only that the process went smoothly.
  }
```

Note that you pass the address of a GUID variable, *guidEffect*, to the **EnumEffects** method. This address is passed in turn to the callback as the *pv* parameter. You also restrict the enumeration to periodic effects by setting the flag DIEFT_PERIODIC.

Step 4: Creating an Effect

If the *EffectFound* flag is no longer FALSE after effect enumeration, you can safely assume that DirectInput has found support for at least one effect of the type you requested. (Of course, in real life you would probably not be content with finding just any periodic effect; you would want to use a particular kind such as a sine or sawtooth.) Armed with the effect GUID, you can now create the effect object.

Before calling the **IDirectInputDevice2::CreateEffect** method, you need to set up the following arrays and structures:

- An array of axes that will be involved in the effect. For a joystick this array will normally consist of the identifiers for the x-axis and the y-axis.
- An array of values for setting the direction. The values will differ according to the number of axes, and according to whether you want to use polar, spherical, or Cartesian coordinates. For a full explanation of this rather complicated business, see Effect Direction.
- A structure of type-specific parameters. In the example, since you are creating a periodic effect, this will be of type **DIPERIODIC**.
- A **DIENVELOPE** structure for defining the envelope to be applied to the effect.
- Finally, a **DIEFFECT** structure to contain the basic parameters for the effect.

First, declare the arrays and structures. You can initialize the arrays at the same time:

```
DWORD      dwAxes[2] = { DIJOFS_X, DIJOFS_Y };
LONG       lDirection[2] = { 0, 0 };

DIPERIODIC diPeriodic;      // type-specific parameters
DIENVELOPE diEnvelope;      // envelope
DIEFFECT   diEffect;        // general parameters
```

Now initialize the type-specific parameters. If you use the values in the example, you will create a full-force periodic effect with a period of one-twentieth of a second.

```
diPeriodic.dwMagnitude = DI_FFNOMINALMAX;
diPeriodic.lOffset = 0;
diPeriodic.dwPhase = 0;
diPeriodic.dwPeriod = (DWORD) (0.05 * DI_SECONDS);
```

To get the effect of the chain-saw motor trying to start, briefly coughing into life, and then slowly dying, you will set an envelope with an attack time of half a second and a fade time of one second. You'll get to the sustain value in a moment.

```
diEnvelope.dwSize = sizeof(DIENVELOPE);
diEnvelope.dwAttackLevel = 0;
diEnvelope.dwAttackTime = (DWORD) (0.5 * DI_SECONDS);
diEnvelope.dwFadeLevel = 0;
diEnvelope.dwFadeTime = (DWORD) (1.0 * DI_SECONDS);
```

Now you set up the basic effect parameters. These include flags to determine how the directions and device objects (buttons and axes) are identified, the sample period and gain for the effect, and pointers to the other data that you have just prepared. You also associate the effect with the fire button of the joystick, so that it will automatically be played whenever that button is pressed.

```
diEffect.dwSize = sizeof(DIEFFECT);
diEffect.dwFlags = DIEFF_POLAR | DIEFF_OBJECTOFFSETS;
diEffect.dwDuration = (DWORD) (2 * DI_SECONDS);
```

```
diEffect.dwSamplePeriod = 0;                // = default
diEffect.dwGain = DI_FFNOMINALMAX;          // no scaling
diEffect.dwTriggerButton = DIJOFS_BUTTON0;
diEffect.dwTriggerRepeatInterval = 0;
diEffect.cAxes = 2;
diEffect.rgdwAxes = &dwAxes[0];
diEffect.rglDirection = &lDirection[0];
diEffect.lpEnvelope = &diEnvelope;
diEffect.cbTypeSpecificParams = sizeof(diPeriodic);
diEffect.lpvTypeSpecificParams = &diPeriodic;
```

So much for the setup. At last you can create the effect:

```
LPDIEFFECT  g_lpdiEffect;  // global effect object

HRESULT hr = g_lpdid2Game->CreateEffect(
                     guidEffect,     // GUID from enumeration
                     &diEffect,      // where the data is
                     &g_lpdiEffect,  // where to put interface pointer
                     NULL);          // no aggregation
if (FAILED(hr))
  {
  OutputDebugString("Failed to create periodic effect");
  }
```

Remember that, by default, the effect is downloaded to the device as soon as it has been created, provided that the device is in an acquired state at the exclusive cooperative level. So if everything has gone according to plan, you should be able to compile, run, press the "fire" button, and feel the sputtering of a chain saw that's out of gas.

Step 5: Playing an Effect

The effect created in the previous step starts in response to the press of a button. In order to create an effect that is to be played in response to an explicit call, you need to go back to Step 4 and modify the **dwTriggerButton** member of the **DIEFFECT** structure, as follows:

```
diEffect.dwTriggerButton = DIEB_NOTRIGGER;
```

Now, suppose you want to make a chain saw that actually starts and keeps going. This is simply a matter of changing the **dwDuration** member as follows:

```
diEffect.dwDuration = INFINITE;
```

Starting the effect is very simple:

```
g_lpdiEffect->Start(1, 0);
```

The effect will keep running until you stop it:

```
g_lpdiEffect->Stop();
```

Note that you don't need to change the envelope you created in the previous step. The attack is played as the effect starts, but the fade value is ignored.

Step 6: Changing an Effect

Your chain saw is merrily rattling away, and now you want to modify the effect to simulate the slowing down of the engine as the saw bites into wood. Fortunately, DirectInput lets you modify the parameters of an effect while it is playing.

To change the effect, you need to set up a **DIEFFECT** structure or have access to the one you used to create the effect. If you are setting up a new structure with local scope, you need to initialize only the **dwSize** member and any members that contain or point to data that is to be changed.

In this case you want to change a type-specific parameter – the period of the effect – so you need to have access to the **DIPERIODIC** structure you used when creating the effect, or else create a local copy with all members initialized. Make sure that the address of the **DIPERIODIC** structure is in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

Now set the new period of the effect:

```
diPeriodic.dwPeriod = (DWORD) (0.08 * DI_SECONDS);
```

Then call the method that actually makes the changes:

```
HRESULT hr = g_lpdiEffect->SetParameters(&diEffect,
                            DIEP_TYPESPECIFICPARAMS)
```

Note the flag that restricts the changes to a single member of the **DIEFFECT** structure.

You can control the way changes are handled by using other flags. For example, by using the DIEP_NODOWNLOAD flag you could change the parameters immediately after starting the effect but delay the implementation until the user actually started cutting wood. Then you would call the **IDirectInputEffect::Download** method. For more information on how to use the various control flags, see **IDirectInputEffect::SetParameters**.

DirectInput Reference

This section contains reference information for the API elements that DirectInput provides. Reference material is divided into the following categories:

- Interfaces
- Functions
- Callback Functions
- Macros
- Structures
- Device Constants
- Return Values

Interfaces

This section contains references for methods of the following DirectInput interfaces:

- **IDirectInput**
- **IDirectInputDevice**
- **IDirectInputEffect**

Note All DirectInput methods have corresponding macros that expand to C or C++ syntax depending on which language is defined. These macros are found in Dinput.h and are not documented separately.

IDirectInput

Applications use the methods of the **IDirectInput** interface to enumerate, create, and retrieve the status of DirectInput devices, initialize the DirectInput object, and invoke an instance of the Windows Control Panel.

The IDirectInput interface is obtained by using the **DirectInputCreate** function.

The methods of the **IDirectInput** interface can be organized into the following groups.

Device Management	**CreateDevice**
	EnumDevices
	GetDeviceStatus
Miscellaneous	**Initialize**
	RunControlPanel

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

AddRef
QueryInterface
Release

The **LPDIRECTINPUT** type is defined as a pointer to the **IDirectInput** interface:

```
typedef struct IDirectInput    *LPDIRECTINPUT;
```

IDirectInput::CreateDevice

The **IDirectInput::CreateDevice** method creates and initializes an instance of a device based on a given GUID.

```
HRESULT CreateDevice(
  REFGUID rguid,
  LPDIRECTINPUTDEVICE *lplpDirectInputDevice,
  LPUNKNOWN pUnkOuter
);
```

Parameters

rguid

Reference to (C++) or address of (C) the instance GUID for the desired input device (see Remarks). The GUID is retrieved through the **IDirectInput::EnumDevices** method, or it can be one of the following predefined GUIDs:

GUID_SysKeyboard	The default system keyboard.
GUID_SysMouse	The default system mouse.

For the above GUID values to be valid, your application must define INITGUID before all other preprocessor directives at the beginning of the source file, or link to DXGUID.LIB.

lplpDirectInputDevice

Address of the **IDirectInputDevice** interface pointer if successful.

pUnkOuter

Address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers will pass NULL.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following:

DIERR_DEVICENOTREG
DIERR_INVALIDPARAM
DIERR_NOINTERFACE
DIERR_NOTINITIALIZED
DIERR_OUTOFMEMORY

Remarks

In C++ the *rguid* parameter must be passed by reference; in C, which does not have pass-by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpdi->CreateDevice(GUID_SysKeyboard, &pdev, NULL);
```

The following shows the same call in C:

```
lpdi->lpVtbl->CreateDevice(lpdi, &GUID_SysKeyboard, &pdev, NULL);
```

Calling this function with *punkOuter* = NULL is equivalent to creating the object via **CoCreateInstance**(*&CLSID_DirectInputDevice, NULL, CLSCTX_INPROC_SERVER, riid, lplpDirectInputDevice*) and then initializing it with **Initialize**.

Calling this function with *punkOuter* != NULL is equivalent to creating the object via **CoCreateInstance**(*&CLSID_DirectInputDevice, punkOuter, CLSCTX_INPROC_SERVER,*

&IID_IUnknown, lplpDirectInputDevice). The aggregated object must be initialized manually.

IDirectInput::EnumDevices

The **IDirectInput::EnumDevice** method enumerates devices that are either currently attached or could be attached to the computer.

```
HRESULT EnumDevices(
  DWORD dwDevType,
  LPDIENUMCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD dwFlags
);
```

Parameters

dwDevType

Device type filter. If this parameter is zero, all device types are enumerated. Otherwise, it is a DIDEVTYPE_* value (see **DIDEVICEINSTANCE**), indicating the device type that should be enumerated.

lpCallback

Address of a callback function that will be called with a description of each DirectInput device.

pvRef

An application-defined 32-bit value that will be passed to the enumeration callback each time it is called.

dwFlags

Flag value that specifies the scope of the enumeration. This parameter can be one or more of the following values.

DIEDFL_ALLDEVICES

All installed devices will be enumerated. This is the default behavior.

DIEDFL_ATTACHEDONLY

Only attached and installed devices.

DIEDFL_FORCEFEEDBACK

Only devices that support force feedback

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

Keep in mind that all installed devices can be enumerated, even if they are not present. For example, a flight stick may be installed on the system but not currently plugged into the computer. Set the *dwFlags* parameter to indicate whether only attached or all installed devices should be enumerated. If the DIEDFL_ATTACHEDONLY flag is not present, all installed devices will be enumerated.

A preferred device type can be passed as a *dwDevType* filter so that only the devices of that type are enumerated.

The *lpCallback* parameter specifies the address of a callback function of the type documented as **DIEnumDevicesProc**. DirectInput calls this function for every device that is enumerated. In the callback, the device type and friendly name, and the product GUID and friendly name, are given for each device. If a single input device can function as more than one DirectInput device type, it will be returned for each device type it supports. For example, a keyboard with a built-in mouse will be enumerated as a keyboard and as a mouse. The product GUID would be the same for each device, however.

IDirectInput::GetDeviceStatus

The **IDirectInput::GetDeviceStatus** method retrieves the status of a specified device.

```
HRESULT GetDeviceStatus(
  REFGUID rguidInstance
);
```

Parameters

rguidInstance

Instance identifier of the device whose status is being checked.

Return Values

If the method succeeds, the return value is DI_OK if the device is attached to DirectInput, or DI_NOTATTACHED otherwise.

If the method fails, the return value may be one of the following error values:

DIERR_GENERIC

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

IDirectInput::Initialize

The **IDirectInput::Initialize** method initializes a DirectInput object. The **DirectInputCreate** function automatically initializes the DirectInput object device after creating it. Applications normally do not need to call this method.

```
HRESULT Initialize(
  HINSTANCE hinst,
  DWORD dwVersion
);
```

Parameters

hinst

Instance handle to the application or DLL that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that may be necessary for backwards compatibility.

It is an error for a DLL to pass the handle of the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle and not the handle of the web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_BETADIRECTINPUTVERSION
DIERR_OLDDIRECTINPUTVERSION

IDirectInput::RunControlPanel

The **IDirectInput::RunControlPanel** method runs the Windows Control Panel to allow the user to install a new input device or modify configurations.

```
HRESULT RunControlPanel(
  HWND hwndOwner,
  DWORD dwFlags
);
```

Parameters

hwndOwner

Handle to the window to be used as the parent window for the subsequent user interface. If this parameter is NULL, no parent window is used.

dwFlags

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

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

See Also

IDirectInputDevice::RunControlPanel

IDirectInputDevice

Applications use the methods of the **IDirectInputDevice** interface to gain and release access to DirectInput devices, manage device properties and information, set behavior, perform initialization, and invoke a device's control panel. The **IDirectInputDevice2** interface adds force feedback capabilities and support for polled devices.

The **IDirectInputDevice** interface is obtained by using the **IDirectInput::CreateDevice** method. The **IDirectInputDevice2** interface is obtained by calling the **IDirectInputDevice::QueryInterface** method; for an example, see Creating a DirectInput Device.

This section is a reference to the methods of these interfaces.

The methods of the **IDirectInputDevice** interface can be organized into the following groups.

Accessing input devices	**Acquire** **Unacquire**
Device information	**GetCapabilities** **GetDeviceData** **GetDeviceInfo** **GetDeviceState** **SetDataFormat** **SetEventNotification**
Device objects	**EnumObjects** **GetObjectInfo**
Device properties	**GetProperty** **SetProperty**
Setting behavior	**SetCooperativeLevel**
Miscellaneous	**Initialize** **RunControlPanel**

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

AddRef
QueryInterface
Release

The **IDirectInputDevice2** interface supports all the above methods as well as the following additional methods:

CreateEffect
EnumCreatedEffectObjects
EnumEffects
Escape
GetEffectInfo

GetForceFeedbackState
Poll
SendForceFeedbackCommand

The **LPDIRECTINPUTDEVICE** and **LPDIRECTINPUTDEVICE2** types are defined as pointers to the **IDirectInput** interface:

```
typedef struct IDirectInputDevice    *LPDIRECTINPUTDEVICE;
typedef struct IDirectInputDevice2   *LPDIRECTINPUTDEVICE2;
```

IDirectInputDevice::Acquire

The **IDirectInputDevice::Acquire** method obtains access to the input device.

```
HRESULT Acquire();
```

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED
DIERR_OTHERAPPHASPRIO

If the method returns S_FALSE, the device has already been acquired.

Remarks

Before a device can be acquired, a data format must be set by using the **IDirectInputDevice::SetDataFormat** method.

Devices must be acquired before calling the **IDirectInputDevice::GetDeviceState** or **IDirectInputDevice::GetDeviceData** methods for that device.

Device acquisition does not use a reference count. Therefore, if an application calls the **IDirectInputDevice::Acquire** method twice, then calls the **IDirectInputDevice::Unacquire** method once, the device is unacquired.

IDirectInputDevice::EnumObjects

The **IDirectInputDevice::EnumObjects** method enumerates the input and force feedback objects available on a device.

```
HRESULT EnumObjects(
  LPDIENUMDEVICEOBJECTSCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD dwFlags
);
```

Parameters

lpCallback

Address of a callback function that receives DirectInputDevice objects. DirectInput provides a prototype of this function as **DIEnumDeviceObjectsProc**.

pvRef

Reference data (context) for callback.

dwFlags

Flags specifying the types of object to be enumerated. The value may be one or more of the following:

DIDFT_ABSAXIS

An absolute axis.

DIDFT_ALL

All objects.

DIDFT_AXIS

An axis, either absolute or relative.

DIDFT_BUTTON

A push button or a toggle button.

DIDFT_FFACTUATOR

An object that contains a force feedback actuator. In other words, forces may be applied to this object.

DIDFT_FFEFFECTTRIGGER

An object that can be used to trigger force feedback effects.

DIDFT_NODATA

An object that does not generate data. Although no data can be read from it, the object can be used as an output actuator in a force feedback effect (if the DIDFT_FFACTUATOR flag is set).

DIDFT_POV

A point-of-view controller.

DIDFT_PSHBUTTON

A push button. A push button is reported as down when the user presses it and as up when the user releases it.

DIDFT_RELAXIS

A relative axis.

DIDFT_TGLBUTTON

A toggle button. A toggle button is reported as down when the user presses it and remains so until the user presses the button a second time.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

Remarks

The DIDFT_FFACTUATOR and DIDFT_FFEFFECTTRIGGER flags in the **dwFlags** member restrict enumeration to objects that meet all the criteria defined by the included flags. For all the other flags, an object is enumerated if it meets the criterion defined by any included flag in this category. For example, (DIDFT_FFACTUATOR | DIDFT_FFEFFECTTRIGGER) restricts enumeration to force feedback trigger objects, and (DIDFT_FFEFFECTRIGGER | DIDFT_TGLBUTTON | DIDFT_PSHBUTTON) restricts enumeration to buttons of any kind that can be used as effect triggers.

IDirectInputDevice::GetCapabilities

The **IDirectInputDevice::GetCapabilities** method obtains the capabilities of the DirectInputDevice object.

```
HRESULT GetCapabilities(
  LPDIDEVCAPS lpDIDevCaps
);
```

Parameters

lpDIDevCaps

Address of a **DIDEVCAPS** structure to be filled with the device capabilities. The **dwSize** member of this structure must be initialized before calling this method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDEVCAPS_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVCAPS_DX3)**. For more information, see Designing for Previous Versions of DirectInput.

IDirectInputDevice::GetDeviceData

The **IDirectInputDevice::GetDeviceData** method retrieves buffered data from the device.

```
HRESULT GetDeviceData(
  DWORD cbObjectData,
  LPDIDEVICEOBJECTDATA rgdod,
  LPDWORD pdwInOut,
  DWORD dwFlags
);
```

Parameters

cbObjectData
Size of the **DIDEVICEOBJECTDATA** structure, in bytes.

rgdod
Array of **DIDEVICEOBJECTDATA** structures to receive the buffered data. The number of elements in this array must be equal to the value of the *pdwInOut* parameter. If this parameter is NULL, then the buffered data is not stored anywhere, but all other side-effects take place.

pdwInOut
On entry, the number of elements in the array pointed to by the *rgdod* parameter. On exit, the number of elements actually obtained.

dwFlags
Flags that control the manner in which data is obtained. This value may be zero or the following flag.

DIGDD_PEEK
Do not remove the items from the buffer. A subsequent **IDirectInputDevice::GetDeviceData** call will read the same data. Normally, data is removed from the buffer after it is read.

Return Values

If the method succeeds, the return value is DI_OK or DI_BUFFEROVERFLOW.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTACQUIRED
DIERR_NOTBUFFERED
DIERR_NOTINITIALIZED

Remarks

Before device data can be obtained, you must set the data format by using the **IDirectInputDevice::SetDataFormat** method, set the buffer size with **IDirectInputDevice::SetProperty** method, and acquire the device by using the **IDirectInputDevice::Acquire** method.

The following example reads up to ten buffered data elements, removing them from the device buffer as they are read.

```
DIDEVICEOBJECTDATA rgdod[10];
DWORD dwItems = 10;
hres = IDirectInputDevice_GetDeviceData(
    pdid,
    sizeof(DIDEVICEOBJECTDATA),
    rgdod,
    &dwItems,
    0);
if (SUCCEEDED(hres)) {
    // dwItems = number of elements read (could be zero)
    if (hres == DI_BUFFEROVERFLOW) {
    // Buffer had overflowed.
    }
}
```

Your application can flush the buffer and retrieve the number of flushed items by specifying NULL for the *rgdod* parameter and a pointer to a variable containing INFINITE for the *pdwInOut* parameter. The following example illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice_GetDeviceData(
            pdid,
            sizeof(DIDEVICEOBJECTDATA),
            NULL,
            &dwItems,
            0);
if (SUCCEEDED(hres)) {
    // Buffer successfully flushed.
    // dwItems = number of elements flushed
    if (hres == DI_BUFFEROVERFLOW) {
        // Buffer had overflowed.
    }
}
```

Your application can query for the number of elements in the device buffer by setting the *rgdod* parameter to NULL, setting *pdwInOut* to INFINITE and setting *dwFlags* to DIGDD_PEEK. The following code fragment illustrates how this can be done:

```
dwItems = INFINITE;
hres = IDirectInputDevice_GetDeviceData(
            pdid,
            sizeof(DIDEVICEOBJECTDATA),
            NULL,
            &dwItems,
            DIGDD_PEEK);
if (SUCCEEDED(hres)) {
    // dwItems = number of elements in buffer
    if (hres == DI_BUFFEROVERFLOW) {
        // Buffer overflow occurred; not all data
        // were successfully captured.
    }
}
```

To query about whether a buffer overflow has occurred, set the *rgdod* parameter to NULL and the *pdwInOut* parameter to zero. The following example illustrates how this can be done:

```
dwItems = 0;
hres = IDirectInputDevice_GetDeviceData(
            pdid,
            sizeof(DIDEVICEOBJECTDATA),
            NULL,
            &dwItems,
            0);
if (hres == DI_BUFFEROVERFLOW) {
    // Buffer overflow occurred
}
```

See Also

IDirectInputDevice2::Poll

Polling and Events

IDirectInputDevice::GetDeviceInfo

The **IDirectInputDevice::GetDeviceInfo** method obtains information about the device's identity.

```
HRESULT GetDeviceInfo(
  LPDIDEVICEINSTANCE pdidi
);
```

Parameters

pdidi

Address of a **DIDEVICEINSTANCE** structure to be filled with information about the device's identity. An application must initialize the structure's **dwSize** member before calling this method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDEVICEINSTANCE_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVICEINSTANCE_DX3)**. For more information, see Designing for Previous Versions of DirectInput.

IDirectInputDevice::GetDeviceState

The **IDirectInputDevice::GetDeviceState** method retrieves instantaneous data from the device.

```
HRESULT GetDeviceState(
  DWORD cbData,
  LPVOID lpvData
);
```

Parameters

cbData
Size of the buffer in the *lpvData* parameter, in bytes.

lpvData
Address of a structure that receives the current state of the device. The format of the data is established by a prior call to the **IDirectInputDevice::SetDataFormat** method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTACQUIRED
DIERR_NOTINITIALIZED
E_PENDING

Remarks

Before device data can be obtained, set the cooperative level by using the **IDirectInputDevice::SetCooperativeLevel** method, then set the data format by using **IDirectInputDevice::SetDataFormat**, and acquire the device by using the **IDirectInputDevice::Acquire** method.

The four predefined data formats require corresponding device state structures according to the following table:

Data Format	State Structure
c_dfDIMouse	**DIMOUSESTATE**
c_dfDIKeyboard	array of 256 bytes
c_dfDIJoystick	**DIJOYSTATE**
c_dfDIJoystick2	**DIJOYSTATE2**

For example, if you passed the *c_dfDIMouse* format to the **IDirectInputDevice::SetDataFormat** method, you must pass a **DIMOUSESTATE** structure to the **IDirectInputDevice::GetDeviceState** method.

See Also

IDirectInputDevice2::Poll

Polling and Events

IDirectInputDevice::GetObjectInfo

The **IDirectInputDevice::GetObjectInfo** method retrieves information about a device object such as a button or axis.

```
HRESULT GetObjectInfo(
  LPDIDEVICEOBJECTINSTANCE pdidoi,
  DWORD dwObj,
  DWORD dwHow
);
```

Parameters

pdidoi

Address of a **DIDEVICEOBJECTINSTANCE** structure to be filled with information about the object. The structure's **dwSize** member must be initialized before this method is called.

dwObj

Value that identifies the object whose information will be retrieved. The value set for this parameter depends on the value specified in the *dwHow* parameter.

dwHow

Value specifying how the *dwObj* parameter should be interpreted. This value can be one of the following:

Value	Meaning
DIPH_DEVICE	The *dwObj* parameter must be zero.
DIPH_BYOFFSET	The *dwObj* parameter is the offset into the current data format of the object whose information is being accessed.
DIPH_BYID	The *dwObj* parameter is the object type/instance identifier. This identifier is returned in the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned from a previous call to the **IDirectInputDevice::EnumObjects** method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

DIERR_OBJECTNOTFOUND

Remarks

For compatibility with DirectX 3, it is also valid to pass a **DIDEVICEOBJECTINSTANCE_DX3** structure with the **dwSize** member initialized to **sizeof(DIDEVICEOBJECTINSTANCE_DX3)**. For more information, see Designing for Previous Versions of DirectInput.

IDirectInputDevice::GetProperty

The **IDirectInputDevice::GetProperty** method retrieves information about the input device.

```
HRESULT GetProperty(
  REFGUID rguidProp,
  LPDIPROPHEADER pdiph
);
```

Parameters

rguidProp

Identifier of the property to be retrieved. This can be one of the predefined values, or a pointer to a GUID that identifies the property. The following properties are predefined for an input device.

DIPROP_AUTOCENTER

Specifies whether device objects are self-centering. See **IDirectInputDevice::SetProperty** for more information.

DIPROP_AXISMODE

Retrieves the axis mode. The retrieved value (DIPROPAXISMODE_ABS or DIPROPAXISMODE_REL) is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

DIPROP_BUFFERSIZE

Retrieves the input-buffer size. The retrieved value is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

The buffer size determines the amount of data that the buffer can hold between calls to the **IDirectInputDevice::GetDeviceData** method before data is lost. This value may be set to zero to indicate that the application will not be reading buffered data from the device. If the buffer size in the **dwData** member of the **DIPROPDWORD** structure is too large to be supported by the device, the largest possible buffer size is set. To determine whether the requested buffer size was set, retrieve the buffer-size property and compare the result with the value you previously attempted to set.

DIPROP_DEADZONE

Retrieves a value for the dead zone of a joystick, in the range 0 to 10,000, where 0 indicates there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

DIPROP_FFGAIN

Retrieves the gain of the device. See **IDirectInputDevice::SetProperty** for more information.

DIPROP_FFLOAD

Retrieves the memory load for the device. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be DIPH_DEVICE.

The **dwData** member contains a value in the range 0 to 100, indicating the percentage of device memory in use.

DIPROP_GRANULARITY

Retrieves the input granularity. The retrieved value is set in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

Granularity represents the smallest distance the object will report movement. Most axis objects have a granularity of one, meaning that all values are possible. Some axes may have a larger granularity. For example, the wheel axis on a mouse may have a granularity of 20, meaning that all reported changes in position will be multiples of 20. In other words, when the user turns the wheel slowly, the device reports a position of zero, then 20, then 40, and so on.

This is a read-only property; you cannot set its value by calling the **IDirectInputDevice::SetProperty** method.

DIPROP_RANGE

Retrieves the range of values an object can possibly report. The retrieved minimum and maximum values are set in the *lMin* and *lMax* members of the associated **DIPROPRANGE** structure. See the description for the *pdiph* parameter for more information.

For some devices, this is a read-only property; you cannot set its value by calling the **IDirectInputDevice::SetProperty** method.

DIPROP_SATURATION

Retrieves a value for the saturation zones of a joystick, in the range 0 to 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, then the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the deadzone).

pdiph

Address of the **DIPROPHEADER** portion of a larger property-dependent structure that contains the **DIPROPHEADER** structure as a member. When retrieving object range information, this value is the address of the **DIPROPHEADER** structure contained within the **DIPROPRANGE** structure. For any other property, this value is the address of the **DIPROPHEADER** structure contained within the **DIPROPDWORD** structure.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED
DIERR_OBJECTNOTFOUND
DIERR_UNSUPPORTED

Remarks

The following C example illustrates how to obtain the value of the DIPROP_BUFFERSIZE property:

```
DIPROPDWORD dipdw;  // DIPROPDWORD contains a DIPROPHEADER structure.
HRESULT hr;
dipdw.diph.dwSize       = sizeof(DIPROPDWORD);
dipdw.diph.dwHeaderSize = sizeof(DIPROPHEADER);
dipdw.diph.dwObj        = 0; // device property
dipdw.diph.dwHow        = DIPH_DEVICE;

hr = IDirectInputDevice_GetProperty(pdid, DIPROP_BUFFERSIZE, &dipdw.diph);
if (SUCCEEDED(hr)) {
    // The dipdw.dwData member contains the buffer size.
}
```

See Also

IDirectInputDevice::SetProperty

IDirectInputDevice::Initialize

The **IDirectInputDevice::Initialize** method initializes a DirectInputDevice object. The **IDirectInput::CreateDevice** method automatically initializes a device after creating it; applications normally do not need to call this method.

```
HRESULT Initialize(
  HINSTANCE hinst,
  DWORD dwVersion,
  REFGUID rguid
);
```

Parameters

hinst

Instance handle to the application or DLL that is creating the DirectInput device object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that may be necessary for backwards compatibility.

It is an error for a DLL to pass the handle of the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle and not the handle of the web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

rguid

Identifier for the instance of the device for which the interface should be associated. The **IDirectInput::EnumDevices** method can be used to determine which instance GUIDs are supported by the system.

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may be one of the following error values:

DIERR_ACQUIRED

DIERR_DEVICENOTREG

If the method returns S_FALSE, the device had already been initialized with the instance GUID passed in though *rGUID*.

Remarks

If this method fails, the underlying object should be considered to be in an indeterminate state and must be reinitialized before use.

IDirectInputDevice::RunControlPanel

The **IDirectInputDevice::RunControlPanel** method runs the DirectInput control panel associated with this device. If the device does not have a control panel associated with it, the default device control panel is launched.

```
HRESULT RunControlPanel(
  HWND hwndOwner,
  DWORD dwFlags
);
```

Parameters

hwndOwner

Parent window handle. If this parameter is NULL, no parent window is used.

dwFlags

Not currently used. Zero is the only valid value.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

IDirectInputDevice::SetCooperativeLevel

The **IDirectInputDevice::SetCooperativeLevel** method establishes the cooperative level for this instance of the device. The cooperative level determines how this instance of the device interacts with other instances of the device and the rest of the system.

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

Parameters

hwnd

Window handle to be associated with the device. This parameter must be a valid top-level window handle that belongs to the process. The window associated with the device must not be destroyed while it is still active in a DirectInput device.

dwFlags

Flags that describe the cooperative level associated with the device. This parameter can be one of the following values:

DISCL_BACKGROUND

The application requires background access. If background access is granted, the device may be acquired at any time, even when the associated window is not the active window.

DISCL_EXCLUSIVE

The application requires exclusive access. If exclusive access is granted, no other instance of the device may obtain exclusive access to the device while it is acquired. Note, however, non-exclusive access to the device is always permitted, even if another application has obtained exclusive access.

An application that acquires the mouse or keyboard device in exclusive mode should always unacquire the devices when it receives WM_ENTERSIZEMOVE and WM_ENTERMENULOOP messages. Otherwise, the user will not be able to manipulate the menu or move and resize the window.

DISCL_FOREGROUND

The application requires foreground access. If foreground access is granted, the device is automatically unacquired when the associated window moves to the background.

DISCL_NONEXCLUSIVE

The application requires non-exclusive access. Access to the device will not interfere with other applications that are accessing the same device.

Applications must specify either DISCL_FOREGROUND or DISCL_BACKGROUND; it is an error to specify both or neither. Similarly, applications must specify either DISCL_EXCLUSIVE or DISCL_NONEXCLUSIVE.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

Remarks

If the system mouse is acquired in exclusive mode, then the pointer will be removed from the screen until the device is unacquired.

Applications must call this method before acquiring the device by using the **IDirectInputDevice::Acquire** method.

IDirectInputDevice::SetDataFormat

The **IDirectInputDevice::SetDataFormat** method sets the data format for the DirectInput device.

```
HRESULT SetDataFormat(
  LPCDIDATAFORMAT lpdf
);
```

Parameters

lpdf

Address of a structure that describes the format of the data the DirectInputDevice should return. An application can define its own **DIDATAFORMAT** structure or use one of the following predefined global variables:

- *c_dfDIKeyboard*
- *c_dfDIMouse*
- *c_dfDIJoystick*
- *c_dfDIJoystick2*

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_ACQUIRED

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

The data format must be set before the device can be acquired by using the **IDirectInputDevice::Acquire** method. It is necessary to set the data format only once. The data format cannot be changed while the device is acquired.

See Also

IDirectInputDevice::GetDeviceState

IDirectInputDevice::SetEventNotification

The **IDirectInputDevice::SetEventNotification** method sets the event notification status. This method specifies an event that is to be set when the device state changes. It is also used to turn off event notification.

```
HRESULT SetEventNotification(
  HANDLE hEvent
);
```

Parameters

hEvent

Handle to the event that is to be set when the device state changes. DirectInput will use the Win32 **SetEvent** function on the handle when the state of the device changes. If the *hEvent* parameter is NULL, then notification is disabled.

The application may create the handle as either a manual-reset or automatic-reset event by using the Win32 **CreateEvent** function. If the event is created as an automatic-reset event, then the operating system will automatically reset the event once a wait has been satisfied. If the event is created as a manual-reset event, then it is the application's responsibility to call the Win32 **ResetEvent** function to reset it. DirectInput will not call the Win32 **ResetEvent** function for event notification handles. Most applications will create the event as an automatic-reset event.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_ACQUIRED
DIERR_HANDLEEXISTS
DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

Remarks

A device state change is defined as any of the following:

- A change in the position of an axis
- A change in the state (pressed or released) of a button
- A change in the direction of a POV control
- Loss of acquisition

Do not call the Win32 **CloseHandle** function on the event while it has been selected into a DirectInputDevice object. You must call this method with the *hEvent* parameter set to NULL before closing the event handle.

The event notification handle cannot be changed while the device is acquired. If the function is successful, then the application can use the event handle in the same manner as any other Win32 event handle.

The following example checks if the handle is currently set without blocking:

```
dwResult = WaitForSingleObject(hEvent, 0);
if (dwResult == WAIT_OBJECT_0) {
    // Event is set. If the event was created as
    // automatic-reset, then it has also been reset.
}
```

The following example illustrates blocking indefinitely until the event is set. Note that this behavior is strongly discouraged because the thread will not respond to the system until the wait is satisfied. In particular, the thread will not respond to Windows messages.

```
dwResult = WaitForSingleObject(hEvent, INFINITE);
if (dwResult == WAIT_OBJECT_0) {
    // Event has been set. If the event was created
    // as automatic-reset, then it has also been reset.
}
```

The following example illustrates a typical message loop for a message-based application that uses two events:.

```
HANDLE ah[2] = { hEvent1, hEvent2 };

while (TRUE) {

    dwResult = MsgWaitForMultipleObjects(2, ah, FALSE,
                        INFINITE, QS_ALLINPUT);
    switch (dwResult) {
    case WAIT_OBJECT_0:
        // Event 1 has been set. If the event was created as
        // automatic-reset, then it has also been reset.
        ProcessInputEvent1();
        break;

    case WAIT_OBJECT_0 + 1:
        // Event 2 has been set. If the event was created as
        // automatic-reset, then it has also been reset.
        ProcessInputEvent2();
        break;

    case WAIT_OBJECT_0 + 2:
        // A Windows message has arrived. Process
        // messages until there aren't any more.
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if (msg.message == WM_QUIT) {
                goto exitapp;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        break;

    default:
        // Unexpected error.
        Panic();
        break;
    }
}
```

The following example illustrates a typical application loop for a non-message-based application that uses two events:

```
HANDLE ah[2] = { hEvent1, hEvent2 };
```

```
DWORD dwWait = 0;

while (TRUE) {

    dwResult = MsgWaitForMultipleObjects(2, ah, FALSE,
                                         dwWait, QS_ALLINPUT);
    dwWait = 0;

    switch (dwResult) {
    case WAIT_OBJECT_0:
        // Event 1 has been set. If the event was
        // created as automatic-reset, then it has also
        // been reset.
        ProcessInputEvent1();
        break;

    case WAIT_OBJECT_0 + 1:
        // Event 2 has been set. If the event was
        // created as automatic-reset, then it has also
        // been reset.
        ProcessInputEvent2();
        break;

    case WAIT_OBJECT_0 + 2:
        // A Windows message has arrived. Process
        // messages until there aren't any more.
        while(PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)){
            if (msg.message == WM_QUIT) {
                goto exitapp;
            }
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        break;

    default:
        // No input or messages waiting.
        // Do a frame of the game.
        // If the game is idle, then tell the next wait
        // to wait indefinitely for input or a message.
        if (!DoGame()) {
            dwWait = INFINITE;
        }
        break;
    }
}
```

See Also

Polling and Events

IDirectInputDevice::SetProperty

The **IDirectInputDevice::SetProperty** method sets properties that define the device behavior. These properties include input buffer size and axis mode.

```
HRESULT SetProperty(
  REFGUID rguidProp,
  LPCDIPROPHEADER pdiph
);
```

Parameters

rguidProp

Identifier of the property to be set. This can be one of the predefined values, or a pointer to a GUID that identifies the property. The following property values are predefined for an input device.

DIPROP_AUTOCENTER

Specifies whether device objects are self-centering. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be DIPH_DEVICE.

The **dwData** member may be one of the following values:

DIPROPAUTOCENTER_OFF: The device should not automatically center when the user releases the device. An application that uses force-feedback should disable the auto-centering spring before playing effects.

DIPROPAUTOCENTER_ON: The device should automatically center when the user releases the device. For example, in this mode, a joystick would engage the self-centering spring.

Note that the use of force feedback effects may interfere with the auto-centering spring. Some devices disable the auto-centering spring when a force-feedback effect is played.

Not all devices support the auto-center property.

DIPROP_AXISMODE

Sets the axis mode. The value being set (**DIPROPAXISMODE_ABS** or **DIPROPAXISMODE_REL**) must be specified in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be set to DIPH_DEVICE.

DIPROP_BUFFERSIZE

Sets the input-buffer size. The value being set must be specified in the **dwData** member of the associated **DIPROPDWORD** structure. See the description for the *pdiph* parameter for more information.

This setting applies to the entire device, so the **dwHow** member of the associated **DIPROPDWORD** structure must be

set to DIPH_DEVICE.

DIPROP_CALIBRATIONMODE

Allows the application to specify whether DirectInput should retrieve calibrated or uncalibrated data from an axis. By default, DirectInput retrieves calibrated data.

Setting the calibration mode for the entire device is equivalent to setting it for each axis individually.

The **dwData** member of the **DIPROPDWORD** structure may be one of the following values:

DIPROPCALIBRATIONMODE_COOKED: DirectInput should return data after applying calibration information. This is the default mode.

DIPROPCALIBRATIONMODE_RAW: DirectInput should return raw, uncalibrated data. This mode is typically used only by Control Panel-type applications.

Note that setting a device into raw mode causes the dead zone, saturation, and range settings to be ignored.

DIPROP_DEADZONE

Sets the value for the dead zone of a joystick, in the range 0 to 10,000, where 0 indicates there is no dead zone, 5,000 indicates that the dead zone extends over 50 percent of the physical range of the axis on both sides of center, and 10,000 indicates that the entire physical range of the axis is dead. When the axis is within the dead zone, it is reported as being at the center of its range.

This setting can be applied to either the entire device or to a specific axis.

DIPROP_FFGAIN

Sets the gain for the device. This setting applies to the entire device, rather than to any particular object, so the **dwHow** member of the associated **DIPROPDWORD** structure must be **DIPH_DEVICE**.

The **dwData** member contains a gain value that is applied to all effects created on the device. The value is an integer in the range 0 to 10,000, specifying the amount by which effect magnitudes should be scaled for the device. For example, a value of 10,000 indicates that all effect magnitudes are to be taken at face value. A value of 9,000 indicates that all effect magnitudes are to be reduced to 90% of their nominal magnitudes.

Setting a gain value is useful when an application wishes to scale down the strength of all force feedback effects uniformly, based on user preferences.

Unlike other properties, the gain can be set when the device is in an acquired state.

DIPROP_RANGE

Sets the range of values an object can possibly report. The minimum and maximum values are taken from the **lMin** and **lMax** members of the associated **DIPROPRANGE** structure.

For some devices, this is a read-only property.

You cannot set a reverse range; **lMax** must be greater than **lMin**.

DIPROP_SATURATION

Sets the value for the saturation zones of a joystick, in the range 0 to 10,000. The saturation level is the point at which the axis is considered to be at its most extreme position. For example, if the saturation level is set to 9,500, then the axis reaches the extreme of its range when it has moved 95 percent of the physical distance from its center position (or from the deadzone).

This setting can be applied to either the entire device or to a specific axis.

pdiph

Address of the **DIPROPHEADER** structure contained within the **DIPROPDWORD** structure. If setting object range information, this is the address of the **DIPROPHEADER** structure contained within the **DIPROPRANGE** structure.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DI_PROPNOEFFECT

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

DIERR_OBJECTNOTFOUND

DIERR_UNSUPPORTED

Remarks

The buffer size determines the amount of data that the buffer can hold between calls to the **IDirectInputDevice::GetDeviceData** method before data is lost. This value may be set to zero to indicate that the application will not be reading buffered data from the device. If the buffer size in the **dwData** member of the **DIPROPDWORD** structure is too large to be supported by the device, the largest possible buffer size is set. To determine whether the requested buffer size was set, retrieve the buffer-size property and compare the result with the value you previously attempted to set.

See Also

IDirectInputDevice::GetProperty

IDirectInputDevice::Unacquire

The **IDirectInputDevice::Unacquire** method releases access to the device.

```
HRESULT Unacquire();
```

Return Values

The return value is DI_OK is the device was unacquired, or DI_NOEFFECT if the device was not in an acquired state to begin with.

IDirectInputDevice2::CreateEffect

The **IDirectInputDevice2::CreateEffect** method creates and initializes an instance of an effect identified by the effect GUID.

```
HRESULT IDirectInputDevice2::CreateEffect(
  REFGUID rguid,
  LPCDIEFFECT lpeff,
  LPDIRECTINPUTEFFECT * ppdeff,
  LPUNKNOWN punkOuter
);
```

Parameters

rguid

The identity of the effect to be created. This can be a predefined effect GUID, or it can be a GUID obtained from **IDirectInputDevice2::EnumEffects**.

The following effect GUIDs are defined:

GUID_ConstantForce
GUID_RampForce
GUID_Square
GUID_Sine
GUID_Triangle
GUID_SawtoothUp
GUID_SawtoothDown
GUID_Spring
GUID_Damper
GUID_Inertia
GUID_Friction
GUID_CustomForce

lpeff

A **DIEFFECT** structure that provides parameters for the created effect. This parameter is optional. If it is NULL, then the effect object is created without parameters. The application must then call the **IDirectInputEffect::SetParameters** method to set the parameters of the effect before it can download the effect.

ppdeff

Location of the pointer to the **IDirectInputEffect** interface, if successful.

punkOuter

The controlling unknown for COM aggregation. The value is NULL if the interface is not aggregated. Most callers will pass NULL.

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may be one of the following error values:

DIERR_DEVICENOTREG
DIERR_DEVICEFULL
DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

If the return value is S_FALSE, the effect was created and the parameters of the effect were updated, but the effect could not be downloaded because the associated device is not acquired in exclusive mode.

IDirectInputDevice2::EnumCreatedEffectObjects

The **IDirectInputDevice2::EnumCreatedEffectObjects** method enumerates all of the currently created effects for this device. Effects created by **IDirectInputDevice2::CreateEffect** are enumerated.

```
HRESULT IDirectInputDevice2::EnumCreatedEffectObjects(
  LPDIENUMCREATEDEFFECTOBJECTSCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD fl
);
```

Parameters

lpCallback

Address of an application-defined callback function. DirectInput provides the prototype function **DIEnumCreatedEffectObjectsProc**.

pvRef

Reference data (context) for callback.

fl

No flags are currently defined. This parameter must be 0.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

Remarks

The results will be unpredictable if you create or destroy an effect while an enumeration is in progress. However, the callback function can safely release the effect passed to it.

IDirectInputDevice2::EnumEffects

The **IDirectinputDevice2::EnumEffects** method enumerates all of the effects supported by the force feedback system on the device. The enumerated GUIDs may represent predefined effects as well as effects peculiar to the device manufacturer.

```
HRESULT IDirectInputDevice2::EnumEffects(
  LPDIENUMEFFECTSCALLBACK lpCallback,
  LPVOID pvRef,
  DWORD dwEffType
);
```

Parameters

lpCallback

Address of an application-defined callback function. DirectInput provides the prototype function **DIEnumEffectsProc**.

pvRef

A 32-bit application-defined value to be passed to the callback function. This parameter may be any 32-bit value; it is declared as **LPVOID** for convenience.

dwEffType

Effect type filter. Use one of the DIEFT_* values to indicate the effect type to be enumerated, or DIEFT_ALL to enumerate all effect types. For a list of these values, see **DIEFFECTINFO**.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM

DIERR_NOTINITIALIZED

If the callback stops the enumeration prematurely, the enumeration is considered to have succeeded.

Remarks

An application can use the **dwEffType** member of the **DIEFFECTINFO** structure to obtain general information about the effect, such as its type and which envelope and condition parameters are supported by the effect.

In order to exploit an effect to its fullest, you must contact the device manufacturer to obtain information on the semantics of the effect and its effect-specific parameters.

IDirectInputDevice2::Escape

The **IDirectInputDevice2::Escape** method sends a hardware-specific command to the driver.

```
HRESULT IDirectInputDevice2::Escape(
  LPDIEFFESCAPE pesc
);
```

Parameters

pesc

A **DIEFFESCAPE** structure that describes the command to be sent. On success, the **cbOutBuffer** member contains the number of bytes of the output buffer actually used.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_DEVICEFULL

DIERR_NOTINITIALIZED

Other device-specific error codes are also possible. Ask the hardware manufacturer for details.

Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is sending the escape to the correct driver by comparing the value of the **guidFFDriver** member of the **DIDEVICEINSTANCE** structure against the value the application is expecting.

IDirectInputDevice2::GetEffectInfo

The **IDirectInputDevice2::GetEffectInfo** method obtains information about an effect.

```
HRESULT IDirectInputDevice2::GetEffectInfo(
  LPDIEFFECTINFO pdei,
  REFGUID rguid
);
```

Parameters

pdei
A DIEFFECTINFO structure that receives information about the effect. The caller must initialize the **dwSize** member of the structure before calling this method.

rguid
Identifier of the effect for which information is being requested.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_DEVICENOTREG
DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

Remarks

In C++ the *rguid* parameter must be passed by reference; in C, which does not have pass-by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpdev2->GetEffectInfo(&dei, GUID_Effect);
```

The following shows the same call in C:

```
lpdev2->lpVtbl->GetEffectInfo(lpdev2, &dei, &GUID_Effect);
```

IDirectInputDevice2::GetForceFeedbackState

The **IDirectInputDevice2::GetForceFeedbackState** method retrieves the state of the device's force feedback system.

```
HRESULT IDirectInputDevice2::GetForceFeedbackState(
  LPDWORD pdwOut
);
```

Parameters

pdwOut

Location for flags that describe the current state of the device's force feedback system.

The value is a combination of the following constants:

DIGFFS_ACTUATORSOFF

The device's force feedback actuators are disabled.

DIGFFS_ACTUATORSON

The device's force feedback actuators are enabled.

DIGFFS_DEVICELOST

The device suffered an unexpected failure and is in an indeterminate state. It must be reset either by unacquiring and reacquiring the device, or by sending a DISFFC_RESET command.

DIGFFS_EMPTY

The device has no downloaded effects.

DIGFFS_PAUSED

Playback of all active effects has been paused.

DIGFFS_POWEROFF

The force feedback system is not currently available. If the device cannot report the power state, then neither DIGFFS_POWERON nor DIGFFS_POWEROFF will be returned.

DIGFFS_POWERON

Power to the force feedback system is currently available. If the device cannot report the power state, then neither DIGFFS_POWERON nor DIGFFS_POWEROFF will be returned.

DIGFFS_SAFETYSWITCHOFF

The safety switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the safety switch, then neither DIGFFS_SAFETYSWITCHON nor DIGFFS_SAFETYSWITCHOFF will be returned.

DIGFFS_SAFETYSWITCHON

The safety switch is currently on, meaning that the device can operate. If the device cannot report the state of the safety switch, then neither DIGFFS_SAFETYSWITCHON nor DIGFFS_SAFETYSWITCHOFF will be returned.

DIGFFS_STOPPED

No effects are playing and the device is not paused.

DIGFFS_USERFFSWITCHOFF

The user force feedback switch is currently off, meaning that the device cannot operate. If the device cannot report the state of the user force feedback switch, then neither DIGFFS_USERFFSWITCHON nor DIGFFS_USERFFSWITCHOFF will be returned.

DIGFFS_USERFFSWITCHON

The user force feedback switch is currently on, meaning that the device can operate. If the device cannot report the state of the user force feedback switch, then neither DIGFFS_USERFFSWITCHON nor DIGFFS_USERFFSWITCHOFF will be returned.

Future versions of DirectInput may define additional flags. Applications should ignore any flags that are not currently defined.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED
DIERR_UNSUPPORTED

Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

IDirectInputDevice2::Poll

The **IDirectInputDevice2::Poll** method retrieves data from polled objects on a DirectInput device. If the device does not require polling, then calling this method has no effect. If a device that requires polling is not polled periodically, no new data will be received from the device. Calling this method causes DirectInput to update the device state, generate input events (if buffered data is enabled), and set notification events (if notification is enabled).

```
HRESULT IDirectInputDevice2::Poll()
```

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_NOTACQUIRED
DIERR_NOTINITIALIZED

Remarks

Before a device data can be polled, the data format must be set by using the **IDirectInputDevice::SetDataFormat** method, and the device must be acquired by using the **IDirectInputDevice::Acquire** method.

See Also

Polling and Events

IDirectInputDevice2::SendForceFeedbackCommand

The **IDirectInputDevice2::SendForceFeedbackCommand** method sends a command to the device's force feedback system.

```
HRESULT IDirectInputDevice2::SendForceFeedbackCommand(
  DWORD dwFlags
);
```

Parameters

dwFlags

A single value indicating the desired change in state. The value may be one of the following:

DISFFC_CONTINUE

Paused playback of all active effects is to be continued. It is an error to send this command when the device is not in a paused state.

DISFFC_PAUSE

Playback of all active effects is to be paused. This command also stops the clock on effects, so that they continue playing to their full duration when restarted.

While the device is paused, new effects may not be started and existing ones may not be modified. Doing so may result in the subsequent DISFFC_CONTINUE command failing to perform properly.

To abandon a pause and stop all effects, use the DISFFC_STOPALL or DISFCC_RESET commands.

DISFFC_RESET

The device's force feedback system is to be put in its startup state. All effects are removed from the device, are no longer valid, and must be recreated if they are to be used again. The device's actuators are disabled.

DISFFC_SETACTUATORSOFF

The device's force feedback actuators are to be disabled. While the actuators are off, effects continue to play but are ignored by the device. Using the analogy of a sound playback device, they are muted rather than paused.

DISFFC_SETACTUATORSON

The device's force feedback actuators are to be enabled.

DISFFC_STOPALL

Playback of any active effects is to be stopped. All active effects will be reset, but are still being maintained by the device and are still valid. If the device is in a paused state, that state is lost.

This command is equivalent to calling the **IDirectInputEffect::Stop** method for each effect playing.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED
DIERR_UNSUPPORTED

Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

IDirectInputEffect

Applications use the methods of the **IDirectInputEffect** interface to manage effects of force feedback devices.

The interface is obtained by using the **IDirectInputDevice2::CreateEffect** method.

The methods of the **IDirectInputEffect** interface can be organized into the following groups.

Effect information	**GetEffectGuid**
	GetEffectStatus
	GetParameters
Effect manipulation	**Download**
	Initialize
	SetParameters
	Start
	Stop
	Unload
Miscellaneous	**Escape**

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

AddRef
QueryInterface
Release

the **LPDIRECTINPUTEFFECT** type is defined as a pointer to the **IDirectInputEffect** interface:

```
typedef struct IDirectInputEffect    *LPDIRECTINPUTEFFECT;
```

IDirectInputEffect::Download

The **IDirectinputEffect::Download** method places the effect on the device. If the effect is already on the device, then the existing effect is updated to match the values set by the **IDirectInputEffect::SetParameters** method.

```
HRESULT IDirectInputEffect::Download(void);
```

Return Values

If the method succeeds, the return value is DI_OK or S_FALSE.

If the method fails, the return value may be one of the following error values:

DIERR_NOTINITIALIZED
DIERR_DEVICEFULL
DIERR_INCOMPLETEEFFECT
DIERR_INPUTLOST
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_INVALIDPARAM
DIERR_EFFECTPLAYING

If the method returns S_FALSE, the effect has already been downloaded to the device.

Remarks

It is valid to update an effect while it is playing. The semantics of such an operation are explained in the reference for **IDirectInputEffect::SetParameters**.

IDirectInputEffect::Escape

The **IDirectInputEffect::Escape** method sends a hardware-specific command to the driver.

```
HRESULT IDirectInputEffect::Escape(
  LPDIEFFESCAPE pesc
);
```

Parameters

pesc

A **DIEFFESCAPE** structure that describes the command to be sent. On success, the **cbOutBuffer** member contains the number of bytes of the output buffer actually used.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_NOTINITIALIZED

DIERR_DEVICEFULL

Other device-specific error codes are also possible. Ask the hardware manufacturer for details.

Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is sending the escape to the correct driver by comparing the value of the **guidFFDriver** member of the **DIDEVICEINSTANCE** structure against the value the application is expecting.

IDirectInputEffect::GetEffectGuid

The **IDirectInputEffect::GetEffectGuid** method retrieves the GUID for the effect represented by the **IDirectInputEffect** object.

```
HRESULT IDirectInputEffect::GetEffectGuid(
  LPGUID pguid
);
```

Parameters

pguid
A **GUID** structure that is filled by the method.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

Remarks

Additional information about the effect can be obtained by passing the GUID to **IDirectInputDevice2::GetEffectInfo**.

IDirectInputEffect::GetEffectStatus

The **IDirectInputEffect::GetEffectStatus** method retrieves the status of an effect.

```
HRESULT IDirectInputEffect::GetEffectStatus(
  LPDWORD pdwFlags
);
```

Parameters

pdwFlags

Status flags for the effect. The value may be zero, or one or more of the following constants:

DIEGES_PLAYING	The effect is playing.
DIEGES_EMULATED	The effect is emulated.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_NOTINITIALIZED

IDirectInputEffect::GetParameters

The **IDirectInputEffect::GetParameters** method retrieves information about an effect.

```
HRESULT IDirectInputEffect::GetParameters(
  LPDIEFFECT peff,
  DWORD dwFlags
);
```

Parameters

peff

Address of a **DIEFFECT** structure that receives effect information. The **dwSize** member must be filled in by the application before calling this function.

dwFlags

Flags specifying which portions of the effect information is to be retrieved. The value may be zero, or one or more of the following constants:

DIEP_ALLPARAMS

The union of all other **DIEP_*** flags, indicating that all members of the **DIEFFECT** structure are being requested.

DIEP_AXES

The **cAxes** and **rgdwAxes** members should receive data. The **cAxes** member on entry contains the sizes (in **DWORD**s) of the buffer pointed to by the **rgdwAxes** member. If the buffer is too small, then the method returns DIERR_MOREDATA and sets **cAxes** to the necessary size of the buffer.

DIEP_DIRECTION

The **cAxes** and **rglDirection** members should receive data. The **cAxes** member on entry contains the size (in **DWORD**s) of the buffer pointed to by the **rglDirection** member. If the buffer is too small, then the **GetParameters** method returns DIERR_MOREDATA and sets **cAxes** to the necessary size of the buffer.

The **dwFlags** member must include at least one of the coordinate system flags (DIEFF_CARTESIAN, DIEFF_POLAR, or DIEFF_SPHERICAL). DirectInput will return the direction of the effect in one of the coordinate systems you specified, converting between coordinate systems as necessary. On exit, exactly one of the coordinate system flags will be set in the **dwFlags** member, indicating which coordinate system DirectInput used. In particular, passing all three coordinate system flags will retrieve the coordinates in exactly the same format in which they were set.

DIEP_DURATION

The **dwDuration** member should receive data.

DIEP_ENVELOPE

The **lpEnvelope** member points to a **DIENVELOPE** structure that should receive data. If the effect does not have an envelope associated with it, then the **lpEnvelope** member will

be set to NULL.

DIEP_GAIN

The **dwGain** member should receive data.

DIEP_SAMPLEPERIOD

The **dwSamplePeriod** member should receive data.

DIEP_TRIGGERBUTTON

The **dwTriggerButton** member should receive data.

DIEP_TRIGGERREPEATINTERVAL

The **dwTriggerRepeatInterval** member should receive data.

DIEP_TYPESPECIFICPARAMS

The **lpvTypeSpecificParams** member points to a buffer whose size is specified by the **cbTypeSpecificParams** member. On return, the buffer will be filled in with the type-specific data associated with the effect, and the **cbTypeSpecificParams** member will contain the number of bytes copied. If the buffer supplied by the application is too small to contain all the type-specific data, then the method returns DIERR_MOREDATA, and the **cbTypeSpecificParams** member will contain the required size of the buffer in bytes.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_MOREDATA
DIERR_NOTINITIALIZED

Remarks

Common errors resulting in a **DIERR_INVALIDPARAM** error include not setting the **dwSize** member of the **DIEFFECT** structure, passing invalid flags, or not setting up the members in the **DIEFFECT** structure properly in preparation for receiving the effect information. For example, if information is to be retrieved in the **dwTriggerButton** member, the **dwFlags** member must be set to either DIEFF_OBJECTIDS or DIEFF_OBJECTOFFSETS, so that DirectInput knows how to describe the button.

IDirectInputEffect::Initialize

The **IDirectInputEffect::Initialize** method initializes a DirectInputEffect object.

```
HRESULT IDirectInputEffect::Initialize(
  HINSTANCE hinst,
  DWORD dwVersion,
  REFGUID rguid
);
```

Parameters

hinst

Instance handle to the application or DLL that is creating the DirectInputEffect object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that may be necessary for backwards compatibility. It is an error for a DLL to pass the handle of the parent application.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

rguid

Identifier of the effect with which the interface is associated. The **IDirectInputDevice2::EnumEffects** method can be used to determine which effect GUIDs are supported by the device.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be DIERR_DEVICENOTREG.

Remarks

If this method fails, the underlying object should be considered to be an indeterminate state and needs to be reinitialized before it can be subsequently used.

The **IDirectInputDevice2::CreateEffect** method automatically initializes the effect after creating it. Applications normally do not need to call the **Initialize** method.

In C++ the *rguid* parameter must be passed by reference; in C, which does not have pass-by-reference, it must be passed by address. The following is an example of a C++ call:

```
lpeff->Initialize(g_hinstDll, DIRECTINPUT_VERSION, GUID_Effect);
```

The following shows the same call in C:

```
lpeff->lpVtbl->Initialize(lpeff, g_hinstDll,
                          DIRECTINPUT_VERSION, &GUID_Effect);
```

IDirectInputEffect::SetParameters

The **IDirectInputEffect::SetParameters** method sets information about an effect.

```
HRESULT IDirectInputEffect::SetParameters(
  LPCDIEFFECT peff,
  DWORD dwFlags
);
```

Parameters

peff

A **DIEFFECT** structure that contains effect information. The **dwSize** member must be filled in by the application before calling this function, as well as any members specified by corresponding bits in the *dwFlags* parameter.

dwFlags

Flags specifying which portions of the effect information are to be set and how the downloading of the parameters should be handled. The value may be zero, or one or more of the following constants:

DIEP_ALLPARAMS

The union of all other **DIEP_*** flags, indicating that all members of the **DIEFFECT** structure are valid.

DIEP_AXES

The **cAxes** and **rgdwAxes** members contain data.

DIEP_DIRECTION

The **cAxes** and **rglDirection** members contain data. The **dwFlags** member specifies (with **DIEFF_CARTESIAN** or **DIEFF_POLAR**) the coordinate system in which the values should be interpreted.

DIEP_DURATION

The **dwDuration** member contains data.

DIEP_ENVELOPE

The **lpEnvelope** member points to a **DIENVELOPE** structure that contains data. To detach any existing envelope from the effect, pass this flag and set the **lpEnvelope** member to NULL.

DIEP_GAIN

The **dwGain** member contains data.

DIEP_NODOWNLOAD

Suppress the automatic **IDirectInputEffect::Download** that is normally performed after the parameters are updated. See **Remarks**.

DIEP_NORESTART

Suppress the stopping and restarting of the effect in order to change parameters. See **Remarks**.

DIEP_SAMPLEPERIOD

The **dwSamplePeriod** member contains data.

DIEP_START

The effect is to be started (or restarted if it is currently playing)

after the parameters are updated. By default, the play state of the effect is not altered.

DIEP_TRIGGERBUTTON

The **dwTriggerButton** member contains data.

DIEP_TRIGGERDELAY

The **dwTriggerDelay** member contains data.

DIEP_TRIGGERREPEATINTERVAL

The **dwTriggerRepeatInterval** member contains data.

DIEP_TYPESPECIFICPARAMS

The **lpvTypeSpecificParams** and **cbTypeSpecificParams** members of the **DIEFFECT** structure contain the address and size of type-specific data for the effect.

Return Values

If the method succeeds, the return value is one of the following:

DI_OK
DI_EFFECTRESTARTED
DI_DOWNLOADSKIPPED
DI_TRUNCATED
DI_TRUNCATEDANDRESTARTED

If the method fails, the return value may be one of the following error values:

DIERR_NOTINITIALIZED
DIERR_INCOMPLETEEFFECT
DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_EFFECTPLAYING

Remarks

The **dwDynamicParams** member of the **DIEFFECTINFO** structure for the effect specifies which parameters can be dynamically updated while the effect is playing.

The **IDirectInputEffect::SetParameters** method automatically downloads the effect, but this behavior can be suppressed by setting the DIEP_NODOWNLOAD flag. If automatic download has been suppressed, then you can manually download the effect by invoking the **IDirectInputEffect::Download** method.

If the effect is playing while the parameters are changed, then the new parameters take effect as if they were the parameters when the effect started.

For example, suppose a periodic effect with a duration of three seconds is started. After two seconds, the direction of the effect is changed. The effect will then continue for one additional second in the new direction. The envelope, phase, amplitude, and other parameters of the effect continue smoothly as if the direction had not changed.

In the same scenario, if after two seconds the duration of the effect were changed to 1.5 seconds, then the effect would stop.

Normally, if the driver cannot update the parameters of a playing effect, the driver is permitted to stop the effect, update the parameters, and then restart the effect. Passing the DIEP_NORESTART flag suppresses this behavior. If the driver cannot update the parameters of an effect while it is playing, the error code DIERR_EFFECTPLAYING is returned and the parameters are not updated.

No more than one of the DIEP_NODOWNLOAD, DIEP_START, and DIEP_NORESTART flags should be set. (It is also valid to pass none of them.)

These three flags control download and playback behavior as follows:

If DIEP_NODOWNLOAD is set, the effect parameters are updated but not downloaded to the device.

If the DIEP_START flag is set, the effect parameters are updated and downloaded to the device, and the effect is started just as if the **IDirectInputEffect::Start** method had been called with the *dwIterations* parameter set to 1 and with no flags. (Combining the update with DIEP_START is slightly faster than calling **Start** separately, because it requires less information to be transmitted to the device.)

If neither DIEP_NODOWNLOAD nor DIEP_START is set and the effect is not playing, then the parameters are updated and downloaded to the device.

If neither DIEP_NODOWNLOAD nor DIEP_START is set and the effect is playing, then the parameters are updated if the device supports on-the-fly updating. Otherwise the behavior depends on the state of the DIEP_NORESTART flag. If it is set, the error code DIERR_EFFECTPLAYING is returned. If it is clear, the effect is stopped, the parameters are updated, and the effect is restarted.

IDirectInputEffect::Start

The **IDirectInputEffect::Start** method begins playing an effect. If the effect is already playing, it is restarted from the beginning. If the effect has not been downloaded or has been modified since its last download, then it will be downloaded before being started. This default behavior can be suppressed by passing the DIES_NODOWNLOAD flag.

```
HRESULT IDirectInputEffect::Start(
  DWORD dwIterations,
  DWORD dwFlags
);
```

Parameters

dwIterations

Number of times to play the effect in sequence. The envelope is re-articulated with each iteration.

To play the effect exactly once, pass 1. To play the effect repeatedly until explicitly stopped, pass INFINITE. To play the effect until explicitly stopped without re-articulating the envelope, modify the effect parameters with the **IDirectInputEffect::SetParameters** method and change its **dwDuration** member to INFINITE.

dwFlags

Flags that describe how the effect should be played by the device. The value may be zero or one or more of the following values:

DIES_SOLO

All other effects on the device should be stopped before the specified effect is played. If this flag is omitted, then the effect is mixed with existing effects already started on the device.

DIES_NODOWNLOAD

Do not automatically download the effect.

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INVALIDPARAM
DIERR_INCOMPLETEEFFECT
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED
DIERR_UNSUPPORTED

Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

Not all devices support multiple iterations.

IDirectInputEffect::Stop

The **IDirectInputEffect::Stop** method stops playing an effect. The parent device must be acquired.

```
HRESULT IDirectInputEffect::Stop(void);
```

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED

Remarks

The device must be acquired at the exclusive cooperative level for this method to succeed.

IDirectInputEffect::Unload

The **IDirectInputEffect::Unload** method removes the effect from the device. If the effect is playing, it is automatically stopped before it is unloaded.

```
HRESULT IDirectInputEffect::Unload(void);
```

Return Values

If the method succeeds, the return value is DI_OK.

If the method fails, the return value may be one of the following error values:

DIERR_INPUTLOST
DIERR_INVALIDPARAM
DIERR_NOTEXCLUSIVEACQUIRED
DIERR_NOTINITIALIZED

Functions

This section is a reference for DirectInput functions other than COM interface methods and callback functions.

The following is the only function that falls into this category:

- **DirectInputCreate** is used to create the DirectInput system.

DirectInputCreate

The **DirectInputCreate** function creates a DirectInput object that supports the *IDirectInput* COM interface.

```
HRESULT DirectInputCreate(
  HINSTANCE hinst,
  DWORD dwVersion,
  LPDIRECTINPUT * lplpDirectInput,
  LPUNKNOWN punkOuter
);
```

Parameters

hinst

Instance handle to the application or DLL that is creating the DirectInput object. DirectInput uses this value to determine whether the application or DLL has been certified and to establish any special behaviors that may be necessary for backwards compatibility.

It is an error for a DLL to pass the handle of the parent application. For example, an ActiveX control embedded in a Web page that uses DirectInput must pass its own instance handle and not the handle of the web browser. This ensures that DirectInput recognizes the control and can enable any special behaviors that may be necessary.

dwVersion

Version number of DirectInput for which the application is designed. This value will normally be DIRECTINPUT_VERSION. Passing the version number of a previous version will cause DirectInput to emulate that version. For more information, see Designing for Previous Versions of DirectInput.

lplpDirectInput

Pointer to an address that will be initialized with a valid **IDirectInput** interface pointer if the call succeeds.

punkOuter

Pointer to the address of the controlling object's **IUnknown** interface for COM aggregation, or NULL if the interface is not aggregated. Most callers will pass NULL. If aggregation is requested, the object returned in **lplpDirectInput* will be a pointer to the **IUnknown** rather than an **IDirectInput** interface, as required by COM aggregation.

Return Values

If the function succeeds, the return value is DI_OK.

If the function fails, the return value may be one of the following error values:

DIERR_BETADIRECTINPUTVERSION

DIERR_INVALIDPARAM

DIERR_OLDDIRECTINPUTVERSION

DIERR_OUTOFMEMORY

Remarks

Calling this function with *punkOuter* = NULL is equivalent to creating the object through **CoCreateInstance(***&CLSID_DirectInput, punkOuter, CLSCTX_INPROC_SERVER, &IID_IDirectInput, lplpDirectInput***),** then initializing it with **Initialize**.

Calling this function with *punkOuter* != NULL is equivalent to creating the object through **CoCreateInstance(***&CLSID_DirectInput, punkOuter, CLSCTX_INPROC_SERVER, &IID_IUnknown, lplpDirectInput***)**. The aggregated object must be initialized manually.

There are separate ANSI and Unicode versions of this service. The ANSI version creates an object that

supports the **IDirectInputA** interface, whereas the Unicode version creates an object that supports the **IDirectInputW** interface. As with other system services that are sensitive to character set issues, macros in the header file map **DirectInputCreate** to the appropriate character set variation.

Callback Functions

The following four functions are prototype callback functions for use with various enumeration methods. Applications can declare one of these callback functions under any name and define it in any way, but the parameter and return types must be the same as in the prototype.

- **DIEnumCreatedEffectObjectsProc**
- **DIEnumDeviceObjectsProc**
- **DIEnumDevicesProc**
- **DIEnumEffectsProc**

DIEnumCreatedEffectObjectsProc

The **DIEnumCreatedEffectObjectsProc** function is an application-defined callback function that receives DirectInputDevice effects as a result of a call to the **IDirectInputDevice2::EnumCreatedEffectObjects** method.

```
BOOL CALLBACK DIEnumCreatedEffectObjectsProc(
  LPDIRECTINPUTEFFECT peff,
  LPVOID pvRef
);
```

Parameters

peff

Pointer to an effect object that has been created.

pvRef

The application-defined value given in the **IDirectInputDevice2::EnumCreatedEffectObjects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumDeviceObjectsProc

The **DIEnumDeviceObjectsProc** function is an application-defined callback function that receives DirectInputDevice objects as a result of a call to the **IDirectInputDevice::EnumObjects** method.

```
BOOL CALLBACK DIEnumDeviceObjectsProc(
  LPCDIDEVICEOBJECTINSTANCE lpddoi,
  LPVOID pvRef
);
```

Parameters

lpddoi

A **DIDEVICEOBJECTINSTANCE** structure that describes the object being enumerated.

pvRef

The application-defined value given in the **IDirectInputDevice::EnumObjects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumDevicesProc

The **DIEnumDevicesProc** function is an application-defined callback function that receives DirectInput devices as a result of a call to the **IDirectInput::EnumDevices** method.

```
BOOL CALLBACK DIEnumDevicesProc(
  LPDIDEVICEINSTANCE lpddi,
  LPVOID pvRef
);
```

Parameters

lpddi

Address of a **DIDEVICEINSTANCE** structure that describes the device instance.

pvRef

The application-defined value given in the **IDirectInput::EnumDevices** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration or DIENUM_STOP to stop the enumeration.

DIEnumEffectsProc

The **DIEnumEffectsProc** function is an application-defined callback function used with the **IDirectInputDevice2::EnumEffects** method.

```
BOOL CALLBACK DIEnumEffectsProc(
  LPCDIEFFECTINFO pdei,
  LPVOID pvRef
);
```

Parameters

pdei
A **DIEFFECTINFO** structure that describes the enumerated effect.

pvRef
Address of application-defined data given to the **IDirectInputDevice2::EnumEffects** method.

Return Values

Returns DIENUM_CONTINUE to continue the enumeration, or DIENUM_STOP to stop it.

Macros

This section describes the following macros used in DirectInput:

- **DIDFT_GETINSTANCE**
- **DIDFT_GETTYPE**
- **DIEFT_GETTYPE**
- **DISEQUENCE_COMPARE**
- **GET_DIDEVICE_SUBTYPE**
- **GET_DIDEVICE_TYPE**

Dinput.h also defines macros for C calls to all the methods of the **IDirectInput** and **IDirectInputDevice** interfaces. These macros eliminate the need for pointers to method tables. For example, here is a C call to the **IDirectInputDevice::Release** method:

```
lpdid->lpVtbl->Release(lpdid));
```

The equivalent macro call looks like this:

```
IDirectInputDevice_Release(lpdid);
```

All these macros take the same parameters as the method calls themselves.

DIDFT_GETINSTANCE

The **DIDFT_GETINSTANCE** macro extracts the object instance number code from a data format type.

```
BYTE DIDFT_GETINSTANCE(
  DWORD dwType
);
```

Parameters

dwType

The DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

DIDFT_GETTYPE

The **DIDFT_GETTYPE** macro extracts the object type code from a data format type.

```
BYTE DIDFT_GETTYPE(
  DWORD dwType
);
```

Parameters

dwType

The DirectInput data format type. The possible values for this parameter are identical to those found in the **dwType** member of the **DIOBJECTDATAFORMAT** structure.

DIEFT_GETTYPE

The **DIEFT_GETTYPE** macro extracts the effect type code from an effect format type.

```
BYTE DIEFT_GETTYPE(
  DWORD dwType
);
```

Parameters

dwType

The DirectInput effect format type. The possible values for this parameter are identical to those found in the **dwEffType** member of the **DIEFFECTINFO** structure.

DISEQUENCE_COMPARE

The **DISEQUENCE_COMPARE** macro compares two DirectInput sequence numbers, compensating for wraparound.

```
BOOL DISEQUENCE_COMPARE(
  DWORD dwSequence1,
  Operator cmp,
  DWORD dwSequence2
);
```

Parameters

dwSequence1
First sequence number to compare.

cmp
One of the following comparison operators: ==, !=, <, >, <=, or >=.

dwSequence2
Second sequence number to compare.

Return Values

Returns a nonzero value if the result of the comparison specified by the *cmp* parameter is true, or zero otherwise.

Remarks

The following example checks whether the *dwSequence1* parameter value precedes the *dwSequence2* parameter value chronologically:

```
BOOL Sooner = (DISEQUENCE_COMPARE(dwSequence1, <, dwSequence2));
```

GET_DIDEVICE_SUBTYPE

The **GET_DIDEVICE_SUBTYPE** macro extracts the device subtype code from a device type description code.

```
BYTE GET_DIDEVICE_SUBTYPE(
  DWORD dwDevType
);
```

Parameters

dwDevType

DirectInput device type description code. The possible values for this parameter are identical to those found in the **dwDevType** member of the **DIDEVICEINSTANCE** structure.

Remarks

The interpretation of the subtype code depends on the primary type.

This macro is defined in Dinput.h as follows:

```
#define GET_DIDEVICE_SUBTYPE(dwDevType) HIBYTE(dwDevType)
```

See Also

GET_DIDEVICE_TYPE, **DIDEVICEINSTANCE**

GET_DIDEVICE_TYPE

The **GET_DIDEVICE_TYPE** macro extracts the device primary type code from a device type description code.

```
BYTE GET_DIDEVICE_TYPE(
  DWORD dwDevType
);
```

Parameters

dwDevType

DirectInput device type description code. Possible values for this parameter are identical to those found in the **dwDevType** member of the **DIDEVICEINSTANCE** structure.

Remarks

This macro is defined in Dinput.h as follows:

```
#define GET_DIDEVICE_TYPE(dwDevType)    LOBYTE(dwDevType)
```

See Also

GET_DIDEVICE_SUBTYPE, **DIDEVICEINSTANCE**

Structures

This section contains information on the following structures used with DirectInput:

- **DICONDITION**
- **DICONSTANTFORCE**
- **DICUSTOMFORCE**
- **DIDATAFORMAT**
- **DIDEVCAPS**
- **DIDEVICEINSTANCE**
- **DIDEVICEOBJECTDATA**
- **DIDEVICEOBJECTINSTANCE**
- **DIEFFECT**
- **DIEFFECTINFO**
- **DIEFFESCAPE**
- **DIENVELOPE**
- **DIJOYSTATE**
- **DIJOYSTATE2**
- **DIMOUSESTATE**
- **DIOBJECTDATAFORMAT**
- **DIPERIODIC**
- **DIPROPDWORD**
- **DIPROPHEADER**
- **DIPROPRANGE**
- **DIRAMPFORCE**

DICONDITION

The **DICONDITION** structure contains type-specific information for effects that are marked as DIEFT_CONDITION.

A pointer to an array of **DICONDITION** structures for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure. The number of elements in the array must be either one, or equal to the number of axes associated with the effect.

```
typedef struct {
    LONG lOffset;
    LONG lPositiveCoefficient;
    LONG lNegativeCoefficient;
    DWORD dwPositiveSaturation;
    DWORD dwNegativeSaturation;
    LONG lDeadBand;
} DICONDITION, *LPDICONDITION;

typedef const DICONDITION *LPCDICONDITION;
```

Members

lOffset
The offset for the condition, in the range -10,000 to +10,000.

lPositiveCoefficient
The coefficient constant on the positive side of the offset, in the range -10,000 to +10,000.

lNegativeCoefficient
The coefficient constant on the negative side of the offset, in the range -10,000 to +10,000.

If the device does not support separate positive and negative coefficients, then the value of **lNegativeCoefficient** is ignored and the value of **lPositiveCoefficient** is used as both the positive and negative coefficients.

dwPositiveSaturation
The maximum force output on the positive side of the offset, in the range 0 to 10,000.

If the device does not support force saturations, then the value of this member is ignored.

lNegativeSaturation
The maximum force output on the negative side of the offset, in the range 0 to 10,000.

If the device does not support force saturations, then the value of this member is ignored.

If the device does not support separate positive and negative saturations, then the value of **lNegativeSaturation** is ignored and the value of **lPositiveSaturation** is used as both the positive and negative saturations.

lDeadBand
The region around **lOffset** where the condition is not active, in the range 0 to 10,000. In other words, the condition is not active between **lOffset** - **lDeadBand** and **lOffset** + **lDeadBand**.

Remarks

Different types of conditions will interpret the parameters differently, but the basic idea is that force resulting from a condition is equal to *A(q - q0)* where *A* is a scaling coefficient, *q* is some metric, and *q0* is the neutral value for that metric.

The simplified formula give above must be adjusted if a nonzero dead band is provided. If the metric is less than **lOffset** - **lDeadBand**, then the resulting force is given by the following formula:

force = **lNegativeCoefficient** * (*q* - (**lOffset** - **lDeadBand**))

Similarly, if the metric is greater than **lOffset** + **lDeadBand**, then the resulting force is given by the

following formula:

force = **lPositiveCoefficient** * (*q* - (**lOffset** + **lDeadBand**))

A spring condition uses axis position as the metric.

A damper condition uses axis velocity as the metric.

An inertia condition uses axis acceleration as the metric.

If the number of **DICONDITION** structures in the array is equal to the number of axes for the effect, then the first structure applies to the first axis, the second applies to the second axis,and so on. For example, a two-axis spring condition with **lOffset** set to zero in both **DICONDITION** structures would have the same effect as the joystick self-centering spring. When a condition is defined for each axis in this way, the effect must not be rotated.

If there is a single **DICONDITION** structure for an effect with more than one axis, then the direction along which the parameters of the **DICONDITION** structure are in effect is determined by the direction parameters passed in the **rglDirection** field of the **DIEFFECT** structure. For example, a friction condition rotated 45 degrees (in polar coordinates) would resist joystick motion in the northeast-southwest direction but would have no effect on joystick motion in the northwest-southeast direction.

DICONSTANTFORCE

The **DICONSTANTFORCE** structure contains type-specific information for effects that are marked as DIEFT_CONSTANTFORCE.

The structure describes a constant force effect.

A pointer to a single **DICONSTANTFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct {
    LONG lMagnitude;
} DICONSTANTFORCE, *LPDICONSTANTFORCE;

typedef const DICONSTANTFORCE *LPCDICONSTANTFORCE;
```

Members

lMagnitude

The magnitude of the effect, in the range -10,000 to +10,000. If an envelope is applied to this effect, then the value represents the magnitude of the sustain. If no envelope is applied, then the value represents the amplitude of the entire effect.

DICUSTOMFORCE

The **DICUSTOMFORCE** structure contains type-specific information for effects that are marked as DIEFT_CUSTOMFORCE.

The structure describes a custom or user-defined force.

A pointer to a **DICUSTOMFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct {
    DWORD cChannels;
    DWORD dwSamplePeriod;
    DWORD cSamples;
    LPLONG rglForceData;
} DICUSTOMFORCE, *LPDICUSTOMFORCE;

typedef const DICUSTOMFORCE *LPCDICUSTOMFORCE;
```

Members

cChannels

The number of channels (axes) affected by this force.

The first channel is applied to the first axis associated with the effect, the second to the second, and so on. If there are fewer channels than axes, then nothing is associated with the extra axes.

If there is but a single channel, then the effect will be rotated in the direction specified by the **rglDirection** member of the **DIEFFECT** structure. If there is more than one channel, then rotation is not allowed.

Not all devices support rotation of custom effects.

dwSamplePeriod

The sample period in microseconds.

cSample

The total number of samples in the **rglForceData**. It must be an integral multiple of the **cChannels**.

rglForceData

Pointer to an array of force values representing the custom force. If multiple channels are provided, then the values are interleaved. For example, if **cChannels** is 3, then the first element of the array belongs to the first channel, the second to the second, and the third to the third.

DIDATAFORMAT

The **DIDATAFORMAT** structure carries information describing a device's data format. This structure is used with the **IDirectInputDevice::SetDataFormat** method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwObjSize;
    DWORD dwFlags;
    DWORD dwDataSize;
    DWORD dwNumObjs;
    LPDIOBJECTDATAFORMAT rgodf;
} DIDATAFORMAT, *LPDIDATAFORMAT;

typedef const DIDATAFORMAT *LPCDIDATAFORMAT;
```

Members

dwSize
Size of this structure, in bytes.

dwObjSize
Size of the **DIOBJECTDATAFORMAT** structure, in bytes.

dwFlags
Flags describing other attributes of the data format. This value can be one of the following:

DIDF_ABSAXIS
The axes are in absolute mode. Setting this flag in the data format is equivalent to manually setting the axis mode property using the **IDirectInputDevice::SetProperty** method. This may not be combined with DIDF_RELAXIS flag.

DIDF_RELAXIS
The axes are in relative mode. Setting this flag in the data format is equivalent to manually setting the axis mode property using the **IDirectInputDevice::SetProperty** method. This may not be combined with the DIDF_ABSAXIS flag.

dwDataSize
Size of a data packet returned by the device, in bytes. This value must be a multiple of 4 and must exceed the largest offset value for an object's data within the data packet.

dwNumObjs
Number of objects in the **rgodf** array.

rgodf
Address to an array of **DIOBJECTDATAFORMAT** structures. Each structure describes how one object's data should be reported in the device data. Typical errors include placing two pieces of information in the same location and placing one piece of information in more than one location.

Remarks

Applications do not typically need to create a **DIDATAFORMAT** structure. An application can use one of the predefined global data format variables, *c_dfDIMouse*, *c_dfDIKeyboard*, *c_dfDIJoystick*, or *c_dfDIJoystick2*.

The following declarations set a data format that can be used by applications that need two axes (reported in absolute coordinates) and two buttons.

```
// Suppose an application uses the following
// structure to read device data.

typedef struct MYDATA {
    LONG  lX;                   // x-axis goes here
    LONG  lY;                   // y-axis goes here
    BYTE  bButtonA;             // One button goes here
    BYTE  bButtonB;             // Another button goes here
    BYTE  bPadding[2];          // Must be dword multiple in size
} MYDATA;

// Then it can use the following data format.

DIOBJECTDATAFORMAT rgodf[ ] = {
  { &GUID_XAxis, FIELD_OFFSET(MYDATA, lX),
    DIDFT_AXIS | DIDFT_ANYINSTANCE, 0, },
  { &GUID_YAxis, FIELD_OFFSET(MYDATA, lY),
    DIDFT_AXIS | DIDFT_ANYINSTANCE, 0, },
  { &GUID_Button, FIELD_OFFSET(MYDATA, bButtonA),
    DIDFT_BUTTON | DIDFT_ANYINSTANCE, 0, },
  { &GUID_Button, FIELD_OFFSET(MYDATA, bButtonB),
    DIDFT_BUTTON | DIDFT_ANYINSTANCE, 0, },
};
#define numObjects (sizeof(rgodf) / sizeof(rgodf[0]))

DIDATAFORMAT df = {
    sizeof(DIDATAFORMAT),       // this structure
    sizeof(DIOBJECTDATAFORMAT), // size of object data format
    DIDF_ABSAXIS,               // absolute axis coordinates
    sizeof(MYDATA),             // device data size
    numObjects,                 // number of objects
    rgodf,                      // and here they are
};
```

DIDEVCAPS

The **DIDEVCAPS** structure contains information about a DirectInput device's capabilities. This structure is used with the **IDirectInputDevice::GetCapabilities** method.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwDevType;
    DWORD dwAxes;
    DWORD dwButtons;
    DWORD dwPOVs;
    DWORD dwFFSamplePeriod;
    DWORD dwFFMinTimeResolution;
    DWORD dwFirmwareRevision;
    DWORD dwHardwareRevision;
    DWORD dwDriverVersion;
} DIDEVCAPS, *LPDIDEVCAPS;
```

Members

dwSize

Size of this structure, in bytes. This member must be initialized by the application before a call to the **IDirectInputDevice::GetCapabilities** method.

dwFlags

Flags associated with the device. This value can be a combination of the following:

DIDC_ATTACHED

The device is physically attached.

DIDC_DEADBAND

The device supports deadband for at least one force feedback condition.

DIDC_EMULATED

Device functionality is emulated.

DIDC_FORCEFEEDBACK

The device supports force feedback.

DIDC_FFFADE

The force feedback system supports the fade parameter for at least one effect. If the device does not support fade then the fade level and fade time parameters of the **DIENVELOPE** structure will be ignored by the device.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual effect will set the DIEFT_FFFADE flag if fade is supported for that effect.

DIDC_FFATTACK

The force feedback system supports the attack envelope parameter for at least one effect. If the device does not support attack then the attack level and attack time parameters of the **DIENVELOPE** structure will be ignored by the device.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual effect will set the DIEFT_FFATTACK flag if attack is supported for that effect.

DIDC_POLLEDDATAFORMAT

At least one object in the current data format is polled rather than interrupt-driven. For these objects, the application must explicitly call the **IDirectInputDevice2::Poll** method in order to obtain data.

DIDC_POLLEDDEVICE

At least one object on the device is polled rather than interrupt-driven. For these objects, the application must explicitly call the **IDirectInputDevice2::Poll** method in order to obtain data.

DIDC_POSNEGCOEFFICIENTS

The force feedback system supports two coefficient values for conditions (one for the positive displacement of the axis and one for the negative displacement of the axis) for at least one condition. If the device does not support both coefficients, then the negative coefficient in the **DICONDITION** structure will be ignored.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_POSNEGCOEFFICIENTS flag if separate positive and negative coefficients are are supported for that condition.

DIDC_POSNEGSATURATION

The force feedback system supports a maximum saturation for both positive and negative force output for at least one condition. If the device does not support both saturation values, then the negative saturation in the **DICONDITION** structure will be ignored.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_POSNEGSATURATION flag if separate positive and negative saturations are are supported for that condition.

DIDC_SATURATION

The force feedback system supports the saturation of condition effects for at least one condition. If the device does not support saturation, then the force generated by a condition is limited only by the maximum force which the device can generate.

After a call to the **IDirectInputDevice2::GetEffectInfo** method, an individual condition will set the DIEFT_SATURATION flag if saturation is supported for that condition.

dwDevType

Device type specifier. This member can contain values identical to those in the **dwDevType** member of the **DIDEVICEINSTANCE** structure.

dwAxes

Number of axes available on the device.

dwButtons

Number of buttons available on the device.

dwPOVs

Number of point-of-view controllers available on the device.

dwFFSamplePeriod

The minimum time between playback of consecutive raw force commands.

dwFFMinTimeResolution

The minimum amount of time, in microseconds, that the device can resolve. The device rounds any times to the nearest supported increment. For example, if the value of **dwFFMinTimeResolution** is 1000, then the device would round any times to the nearest millisecond.

dwFirmwareRevision
Specifies the firmware revision of the device.

dwHardwareRevision
The hardware revision of the device.

dwDriverVersion
The version number of the device driver.

Remarks

The semantics of version numbers are left to the manufacturer of the device. The only guarantee is that newer versions will have larger numbers.

See Also

DIDEVICEINSTANCE

DIDEVICEINSTANCE

The **DIDEVICEINSTANCE** structure contains information about an instance of a DirectInput device. This structure is used with the **IDirectInput::EnumDevices** and **IDirectInputDevice::GetDeviceInfo** methods.

```
typedef struct {
    DWORD dwSize;
    GUID  guidInstance;
    GUID  guidProduct;
    DWORD dwDevType;
    TCHAR tszInstanceName[MAX_PATH];
    TCHAR tszProductName[MAX_PATH];
    GUID guidFFDriver;
    WORD wUsagePage;
    WORD wUsage;
} DIDEVICEINSTANCE, *LPDIDEVICEINSTANCE;

typedef const DIDEVICEINSTANCE  *LPCDIDEVICEINSTANCE;
```

Members

dwSize
Size of this structure, in bytes. This member must be initialized before the structure is used.

guidInstance
Unique identifier for the instance of the device. An application may save the instance GUID into a configuration file and use it at a later time. Instance GUIDs are specific to a particular computer. An instance GUID obtained from one computer is unrelated to instance GUIDs on another.

guidProduct
Unique identifier for the product. This identifier is established by the manufacturer of the device.

dwDevType
Device type specifier. The least-significant byte of the device type description code specifies the device type. The next-significant byte specifies the device subtype. This value can be one of the following types combined with their respective subtypes and optionally with DIDEVTYPE_HID, which specifies a Human Interface Device.

DIDEVTYPE_MOUSE
A mouse or mouse-like device (such as a trackball).

DIDEVTYPE_KEYBOARD
A keyboard or keyboard-like device.

DIDEVTYPE_JOYSTICK
A joystick or similar device, such as a steering wheel

DIDEVTYPE_DEVICE
A device that does not fall into the above categories

The following subtypes are defined for mouse-type devices.

DIDEVTYPEMOUSE_UNKNOWN

The subtype could not be determined.

DIDEVTYPEMOUSE_TRADITIONAL

The device is a traditional mouse.

DIDEVTYPEMOUSE_FINGERSTICK

The device is a fingerstick.

DIDEVTYPEMOUSE_TOUCHPAD

The device is a touchpad.

DIDEVTYPEMOUSE_TRACKBALL

The device is a trackball.

The following subtypes are defined for keyboard-type devices.

DIDEVTYPEKEYBOARD_UNKNOWN

The subtype could not be determined

DIDEVTYPEKEYBOARD_PCXT

IBM PC/XT 83-key keyboard.

DIDEVTYPEKEYBOARD_OLIVETTI

Olivetti 102-key keyboard.

DIDEVTYPEKEYBOARD_PCAT

IBM PC/AT 84-key keyboard.

DIDEVTYPEKEYBOARD_PCENH

IBM PC Enhanced 101/102-key or Microsoft Natural keyboard.

DIDEVTYPEKEYBOARD_NOKIA1050

Nokia 1050 keyboard.

DIDEVTYPEKEYBOARD_NOKIA9140

Nokia 9140 keyboard.

DIDEVTYPEKEYBOARD_NEC98

Japanese NEC PC98 keyboard.

DIDEVTYPEKEYBOARD_NEC98LAPTOP

Japanese NEC PC98 laptop keyboard.

DIDEVTYPEKEYBOARD_NEC98106

Japanese NEC PC98 106-key keyboard.

DIDEVTYPEKEYBOARD_JAPAN106

Japanese 106-key keyboard.

DIDEVTYPEKEYBOARD_JAPANAX

Japanese AX keyboard.

DIDEVTYPEKEYBOARD_J3100

Japanese J3100 keyboard.

The following subtypes are defined for joystick-type devices.

DIDEVTYPEJOY

STICK_UNKNOWN

The subtype could not be determined.

DIDEVTYPEJOYSTICK_TRADITIONAL

A traditional joystick.

DIDEVTYPEJOYSTICK_FLIGHTSTICK

A joystick optimized for flight simulation.

DIDEVTYPEJOYSTICK_GAMEPAD

A device whose primary purpose is to provide button input.

DIDEVTYPEJOYSTICK_RUDDER

A device for yaw control.

DIDEVTYPEJOYSTICK_WHEEL

A steering wheel.

DIDEVTYPEJOYSTICK_HEADTRACKER

A device that tracks the movement of the user's head

The high-order word of the device type description code contains flags that further identify the device.

DIDEVTYPE_HID

The device uses the Human Input Device (HID) protocol.

tszInstanceName[MAX_PATH]
Friendly name for the instance. For example, "Joystick 1."

tszProductName[MAX_PATH]
Friendly name for the product.

guidFFDriver
Unique identifier for the driver being used for force feedback. This identifier is established by the manufacturer of the driver.

wUsagePage
If the device is a HID device, then this member contains the HID usage page code.

wUsage
If the device is a HID device, then this member contains the HID usage code.

Remarks

For compatibility with previous versions of DirectX, a **DIDEVICEINSTANCE_DX3** structure is also defined, containing only the first six members of the **DIDEVICEINSTANCE** structure.

DIDEVICEOBJECTDATA

The **DIDEVICEOBJECTDATA** structure contains raw buffered device information. This structure is used with the **IDirectInputDevice::GetDeviceData** method.

```
typedef struct {
    DWORD dwOfs;
    DWORD dwData;
    DWORD dwTimeStamp;
    DWORD dwSequence;
} DIDEVICEOBJECTDATA, *LPDIDEVICEOBJECTDATA;

typedef const DIDEVICEOBJECTDATA *LPCDIDEVICEOBJECTDATA;
```

Members

dwOfs

Value specifying the offset into the current data format of the object whose data is being reported. That is, the location where the **dwData** would have been stored if the data had been obtained by a call to the **IDirectInputDevice::GetDeviceState** method.

If the device is accessed as a mouse, keyboard, or joystick, the **dwOfs** member will be one of the mouse device constants, keyboard device constants, or joystick device constants. If a custom data format has been set, then it will be an offset relative to the custom data format.

dwData

Data obtained from the device. The format of this data depends on the type of the device, but in all cases, the data is reported in raw form.

For axes, if the device is in relative axis mode, then the relative axis motion is reported. If the device is in absolute axis mode, then the absolute axis coordinate is reported.

For buttons, only the low byte of **dwData** is significant. The high bit of the low byte is set if the button went down; it is clear if the button went up.

dwTimeStamp

Tick count at which the event was generated, in milliseconds. The current system tick count can be obtained by calling the Win32 **GetTickCount** function. Remember that this value wraps around approximately every 50 days.

dwSequence

DirectInput sequence number for this event. All DirectInput events are assigned an increasing sequence number. This allows events from different devices to be sorted chronologically. Since this value can wrap around, care must be taken when comparing two sequence numbers. The **DISEQUENCE_COMPARE** macro can be used to perform this comparison safely.

DIDEVICEOBJECTINSTANCE

The **DIDEVICEOBJECTINSTANCE** structure contains information about a device object instance. This structure is used with the **IDirectInputDevice::EnumObjects** method to provide the **DIEnumDeviceObjectsProc** callback function with information about a particular object associated with a device, like an axis or button. It is also used with the **IDirectInputDevice::GetObjectInfo** method to retrieve information about a device object.

```
typedef struct {
    DWORD dwSize;
    GUID  guidType;
    DWORD dwOfs;
    DWORD dwType;
    DWORD dwFlags;
    TCHAR tszName[MAX_PATH];
    DWORD dwFFMaxForce;
    DWORD dwFFForceResolution;
    WORD wCollectionNumber;
    WORD wDesignatorIndex;
    WORD wUsagePage;
    WORD wUsage;
    DWORD dwDimension;
    WORD wExponent;
    WORD wReserved;
} DIDEVICEOBJECTINSTANCE, *LPDIDEVICEOBJECTINSTANCE;

typedef const DIDEVICEOBJECTINSTANCE *LPCDIDEVICEOBJECTINSTANCE;
```

Members

dwSize

Size of the structure, in bytes. During enumeration, the application may inspect this value to determine how many members of the structure are valid. When the structure is passed to the **IDirectInputDevice::GetObjectInfo** method, this member must be initialized to **sizeof(DIDEVICEOBJECTINSTANCE)**.

guidType

Unique identifier that indicates the object type. This member is optional. If present, it can be one of the following values:

GUID_XAxis

The horizontal axis. For example, it may represent the left-right motion of a mouse.

GUID_YAxis

The vertical axis. For example, it may represent the forward-backward motion of a mouse.

GUID_ZAxis

The z-axis. For example, it may represent rotation of the wheel on a mouse, or movement of a throttle control on a joystick.

GUID_RxAxis

Rotation around the x-axis.

GUID_RyAxis

Rotation around the y-axis.

GUID_RzAxis

Rotation around the z-axis (often a rudder control).

GUID_Slider

A slider axis.

GUID_Button

A button on a mouse.

GUID_Key

A key on a keyboard.

GUID_POV

A point-of-view indicator or “hat”.

GUID_Unknown

Unknown.

Other object types may be defined in the future.

dwOfs

Offset within the data format at which the data reported by this object is most efficiently obtained. This member is significant only for applications that build custom data formats. Most applications will not use this value.

dwType

Device type that describes the object. It is a combination of DIDFT_* flags that describe the object type (axis, button, and so forth) and contains the object instance number in the middle 16 bits. Use the **DIDFT_GETINSTANCE** macro to extract the object instance number. For the DIDFT_* flags, see **IDirectInputDevice::EnumObjects**.

dwFlags

Flags describing other attributes of the data format. This value can be one of the following:

DIDOI_ASPECTACCEL

The object reports acceleration information.

DIDOI_ASPECTFORCE

The object reports force information.

DIDOI_ASPECTMASK

The bits that are used to report aspect information. An object can represent at most one aspect.

DIDOI_ASPECTPOSITION

The object reports position information.

DIDOI_ASPECTVELOCITY

The object reports velocity information.

DIDOI_FFACTUATOR

The object can have force feedback effects applied to it.

DIDOI_FFEFFECTTRIGGER

The object can trigger playback of force feedback effects.

DIDOI_POLLED

The object does not return data until the **IDirectInputDevice2::Poll** method is called.

tszName[MAX_PATH]

Name of the object; for example, "X-Axis" or "Right Shift."

dwFFMaxForce
The magnitude of the maximum force that can be created by the actuator associated with this object. Force is expressed in newtons and measured in relation to where the hand would be during normal operation of the device.

dwFFForceResolution
The force resolution of the actuator associated with this object. The returned value represents the number of gradations, or subdivisions, of the maximum force that can be expressed by the force feedback system from 0 (no force) to maximum force.

wCollectionNumber
Reserved.

wDesignatorIndex
Reserved.

wUsagePage
The HID usage page associated with the object, if known. HID devices will always report a usage page. Non-HID devices may optionally report a usage page; if they do not, then the value of this member will be zero.

wUsage
The HID usage associated with the object, if known. HID devices will always report a usage. Non-HID devices may optionally report a usage; if they do not, then the value of this member will be zero.

dwDimension
The dimensional units in which the object's value is reported, if known, or zero if not known. Applications can use this field to distinguish between, for example, the position and velocity of a control.

wExponent
The exponent to associate with the dimension, if known.

wReserved
Reserved.

Remarks

Applications can use the **wUsagePage** and **wUsage** members to obtain additional information about how the object was designed to be used. For example, if **wUsagePage** has the value 0x02 (vehicle controls) and **wUsage** has the value 0xB9 (elevator trim), then the object was designed to be the elevator trim control on a flightstick. A flight simulator application can use this information to provide more reasonable defaults for objects on the device. HID usage codes are determined by the USB standards committee.

DIEFFECT

The **DIEFFECT** structure is used by the **IDirectInputDevice2::CreateEffect** method to initialize a new **IDirectInputEffect** object. It is also used by the **IDirectInputEffect::SetParameters** and **IDirectInputEffect::GetParameters** methods.

```
typedef struct {
    DWORD dwSize;
    DWORD dwFlags;
    DWORD dwDuration;
    DWORD dwSamplePeriod;
    DWORD dwGain;
    DWORD dwTriggerButton;
    DWORD dwTriggerRepeatInterval;
    DWORD cAxes;
    LPDWORD rgdwAxes;
    LPLONG rglDirection;
    LPDIENVELOPE lpEnvelope;
    DWORD cbTypeSpecificParams;
    LPVOID lpvTypeSpecificParams;
} DIEFFECT, *LPDIEFFECT;

typedef const DIEFFECT *LPCDIEFFECT;
```

Members

dwSize

Specifies the size, in bytes, of the structure. This member must be initialized before the structure is used.

dwFlags

Flags associated with the effect. This value can be a combination of one or more of the following values:

DIEFF_CARTESIAN

The values of **rglDirection** are to be interpreted as Cartesian coordinates.

DIEFF_OBJECTIDS

The values of **dwTriggerButton** and **rgdwAxes** are object identifiers as obtained via **IDirectInputDevice::EnumObjects**.

DIEFF_OBJECTOFFSETS

The values of **dwTriggerButton** and **rgdwAxes** are data format offsets, relative to the data format selected by **IDirectInput::SetDataFormat**.

DIEFF_POLAR

The values of **rglDirection** are to be interpreted as polar coordinates.

DIEFF_SPHERICAL

The values of **rglDirection** are to be interpreted as spherical coordinates.

dwDuration

The total duration of the effect in microseconds. If this value is INFINITE, then the effect has infinite duration. If an envelope has been applied to the effect, then the attack will be applied, followed by an infinite sustain.

dwSamplePeriod

The period at which the device should play back the effect, in microseconds. A value of zero indicates that the default playback sample rate should be used.

If the device is not capable of playing back the effect at the specified rate, it will choose the supported rate that is closest to the requested value.

Setting a custom **dwSamplePeriod** can be used for special effects. For example, playing a sine wave at an artificially large sample period results in a rougher texture.

dwGain

The gain to be applied to the effect, in the range 0 to 10,000. The gain is a scaling factor applied to all magnitudes of the effect and its envelope.

dwTriggerButton

The identifier or offset of the button to be used to trigger playback of the effect. The flags DIEFF_OBJECTIDS and DIEFF_OBJECTOFFSETS determine the semantics of the value. If this member is set to DIEB_NOTRIGGER, then no trigger button is associated with the effect.

dwTriggerRepeatInterval

The interval, in microseconds, between the end of one playback and the start of the next when the effect is triggered by a button press and the button is held down. Setting this value to INFINITE suppresses repetition.

Support for trigger repeat for an effect is indicated by the presence of the DIEP_TRIGGERREPEATINTERVAL flag in the **dwStaticParams** member of the **DIEFFECTINFO** structure.

cAxes

Number of axes involved in the effect. This member must be filled in by the caller if changing or setting the axis list or the direction list.

The number of axes for an effect cannot be changed once it has been set.

rgdwAxes

Pointer to a **DWORD** array (of **cAxes** elements) containing identifiers or offsets identifying the axes to which the effect is to be applied. The flags DIEFF_OBJECTIDS and DIEFF_OBJECTOFFSETS determine the semantics of the values in the array.

The list of axes associated with an effect cannot be changed once it has been set.

No more than 32 axes can be associated with a single effect.

rglDirection

Pointer to a **LONG** array (of **cAxes** elements) containing either Cartesian coordinates or polar coordinates. The flags DIEFF_CARTESIAN. DIEFF_POLAR, and DIEFF_SPHERICAL determine the semantics of the values in the array.

If Cartesian, then each value in **rglDirection** is associated with the corresponding axis in **rgdwAxes**.

If polar, then the angle is measured in hundredths of degrees from the (0, -1) direction, rotated in the direction of (1, 0). This usually means that "north" is away from the user, and "east" is to the user's right. The last element is not used.

If spherical, then the first angle is measured in hundredths of degrees from the (1, 0) direction, rotated in the direction of (0, 1). The second angle (if the number of axes is three or more) is measured in hundredths of degrees towards (0, 0, 1). The third angle (if the number of axes is four or more) is measured in hundredths of degrees towards (0, 0, 0, 1), and so on. The last element is not used.

Note The **rglDirection** array must contain **cAxes** entries, even if polar or spherical coordinates are given. In these cases the last element in the **rglDirection** array is reserved for future use and

must be zero.

lpEnvelope

Optional pointer to a **DIENVELOPE** structure that describes the envelope to be used by this effect. Note that not all effect types use envelopes. If no envelope is to be applied, then the member should be set to NULL.

cbTypeSpecificParams

Number of bytes of additional type-specific parameters for the corresponding effect type.

lpvTypeSpecificParams

Pointer to type-specific parameters, or NULL if there are no type-specific parameters.

If the effect is of type DIEFT_CONDITION, then this member contains a pointer to an array of **DICONDITION** structures that define the parameters for the condition. A single structure may be used, in which case the condition is applied in the direction specified in the **rglDirection** array. Otherwise there must be one structure for each axis, in the same order as the axes in **rgdwAxes** array. If a structure is supplied for each axis, the effect should not be rotated; you should use the following values in the **rglDirection** array:

- DIEFF_SPHERICAL: 0, 0, ...
- DIEFF_POLAR: 9000, 0, ...
- DIEFF_CARTESIAN: 1, 0, ...)

If the effect is of type DIEFT_CUSTOMFORCE, then this member contains a pointer to a **DICUSTOMFORCE** structure that defines the parameters for the custom force.

If the effect is of type DIEFT_PERIODIC, then this member contains a pointer to a **DIPERIODIC** structure that defines the parameters for the effect.

If the effect is of type DIEFT_CONSTANTFORCE, then this member contains a pointer to a **DICONSTANTFORCE** structure that defines the parameters for the constant force.

If the effect is of type DIEFT_RAMPFORCE, then this member contains a pointer to a **DIRAMPFORCE** structure that defines the parameters for the ramp force.

DIEFFECTINFO

The **DIEFFECTINFO** structure is used by the **IDirectInputDevice2::EnumEffects** and **IDirectInputDevice2::GetEffectInfo** methods to return information about a particular effect supported by a device.

```
typedef struct {
    DWORD dwSize;
    GUID guid;
    DWORD dwEffType;
    DWORD dwStaticParams;
    DWORD dwDynamicParams;
    TCHAR tszName[MAX_PATH];
} DIEFFECTINFO, *LPDIEFFECTINFO;

typedef const DIEFFECTINFO  *LPCDIEFFECTINFO;
```

Members

dwSize

The size of the structure in bytes. During enumeration, the application may inspect this value to determine how many members of the structure are valid. This member must be initialized before the structure is passed to the **IDirectInputDevice2::GetEffectInfo** method.

guid

Identifier of the effect.

dwEffType

Zero or more of the following values:

DIEFT_ALL

Valid only for **IDirectInputDevice2::EnumEffects.** Enumerate all effects, regardless of type. This flag may not be combined with any of the other flags.

DIEFT_CONDITION

The effect represents a condition. When creating or modifying a condition, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to an array of **DICONDITION** structures (one per axis) and the **cbTypeSpecificParams** member must be set to **cAxis** * **sizeof(DICONDITION)**.

Not all devices support all the parameters of conditions. Check the effect capability flags to determine which capabilities are available.

The flag can be passed to **IDirectInputDevice2::EnumEffects** to restrict the enumeration to conditions.

DIEFT_CONSTANTFORCE

The effect represents a constant-force effect. When creating or modifying a constant-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** must point to a **DICONSTANTFORCE** structure and the **cbTypeSpecificParams** member must be set to sizeof(**DICONSTANTFORCE**).

The flag can be passed to **IDirectInputDevice2::EnumEffects** to restrict the enumeration to constant-force effects.

DIEFT_CUSTOMFORCE

The effect represents a custom-force effect. When creating or modifying a custom-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DICUSTOMFORCE** structure and the **cbTypeSpecificParams** member must be set to sizeof(**DICUSTOMFORCE**).

The flag can be passed to **IDirectInputDevice2::EnumEffects** to restrict the enumeration to custom-force effects.

DIEFT_DEADBAND

The effect generator for this condition effect supports the **lDeadBand** parameter.

DIEFT_FFATTACK

The effect generator for this effect supports the attack envelope parameter. If the effect generator does not support attack then the attack level and attack time parameters of the **DIENVELOPE** structure will be ignored by the effect.

If neither DIEFT_FFATTACK nor DIEFT_FFFADE is set, then the effect does not support an envelope, and any provided envelope will be ignored.

DIEFT_FFFADE

The effect generator for this effect supports the fade parameter. If the effect generator does not support fade then the fade level and fade time parameters of the **DIENVELOPE** structure will be ignored by the effect.

If neither DIEFT_FFATTACK nor DIEFT_FFFADE is set, then the effect does not support an envelope, and any provided envelope will be ignored.

DIEFT_HARDWARE

The effect represents a hardware-specific effect. For additional information on using a hardware-specific effect, consult the hardware documentation.

The flag can be passed to the **IDirectInputDevice2::EnumEffects** method to restrict the enumeration to hardware-specific effects.

DIEFT_PERIODIC

The effect represents a periodic effect. When creating or modifying a periodic effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DIPERIODIC** structure and the **cbTypeSpecificParams** member must be set to sizeof(**DIPERIODIC**).

The flag can be passed to **IDirectInputDevice2::EnumEffects** to restrict the enumeration to periodic effects.

DIEFT_POSNEGCOEFFICIENTS

The effect generator for this effect supports two coefficient values for conditions, one for the positive displacement of the axis and one for the negative displacement of the axis. If the device does not support both coefficients, then the negative coefficient in the **DICONDITION** structure will be ignored and the positive coefficient will be used in both directions.

DIEFT_POSNEGSATURATION

The effect generator for this effect supports a maximum

saturation for both positive and negative force output. If the device does not support both saturation values, then the negative saturation in the **DICONDITION** structure will be ignored and the positive saturation will be used in both directions.

DIEFT_RAMPFORCE

The effect represents a ramp-force effect. When creating or modifying a ramp-force effect, the **lpvTypeSpecificParams** member of the **DIEFFECT** structure must point to a **DIRAMPFORCE** structure and the **cbTypeSpecificParams** member must be set to sizeof(**DIRAMPFORCE**).

The flag can be passed to **IDirectInputDevice2::EnumEffects** to restrict the enumeration to ramp-force effects.

DIEFT_SATURATION

The effect generator for this effect supports the saturation of condition effects. If the effect generator does not support saturation, then the force generated by a condition is limited only by the maximum force that the device can generate.

dwStaticParams

Zero or more DIEP_* values describing the parameters supported by the effect. For example, if DIEP_ENVELOPE is set, then the effect supports an envelope. For a list of possible values, see **IDirectInputEffect::GetParameters**.

It is not an error for an application to attempt to use effect parameters which are not supported by the device. The unsupported parameters are merely ignored.

This information is provided to allow the application to tailor its use of force feedback to the capabilities of the specific device.

dwDynamicParams

Zero or more **DIEP_*** values denoting parameters of the effect that can be modified while the effect is playing. For a list of possible values, see **IDirectInputEffect::GetParameters**.

If an application attempts to change a parameter while the effect is playing, and the driver does not support modifying that effect dynamically, then driver is permitted to stop the effect, update the parameters, then restart it. See **IDirectInputEffect::SetParameters** for more information.

tszName[MAX_PATH]

Name of the effect; for example, "Sawtooth up" or "Constant force".

Remarks

Use the **DIEFT_GETTYPE** macro to extract the effect type from the **dwEffType** flags.

DIEFFESCAPE

The **DIEFFESCAPE** structure is used by the **IDirectInputEffect::Escape** method to pass hardware-specific data directly to the device driver.

```
typedef struct {
    DWORD dwSize;
    DWORD dwCommand;
    LPVOID lpvInBuffer;
    DWORD cbInBuffer;
    LPVOID lpvOutBuffer;
    DWORD cbOutBuffer;
} DIEFFESCAPE, *LPDIEFFESCAPE;
```

Members

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

dwCommand
Driver-specific command number. Consult the driver documentation for a list of valid commands.

lpvInBuffer
Buffer containing the data required to perform the operation.

cbInBuffer
The size, in bytes, of the **lpvInBuffer** buffer.

lpvOutBuffer
Buffer in which the operation's output data is returned.

cbOutBuffer
On entry, the size in bytes of the **lpvOutBuffer** buffer. On exit, the number of bytes actually produced by the command.

Remarks

Since each driver implements different escapes, it is the application's responsibility to ensure that it is talking to the correct driver by comparing the **guidFFDriver** member in the **DIDEVICEINSTANCE** structure against the value the application is expecting.

DIENVELOPE

The **DIENVELOPE** structure is used by the **DIEFFECT** structure to specify the optional envelope parameters for an effect. The sustain level for the envelope is represented by the **dwMagnitude** member of the **DIPERIODIC** structure and the **lMagnitude** member of the **DICONSTANTFORCE** structure. The sustain time is represented by **dwDuration** member of the **DIEFFECT** structure.

```
typedef struct {
    DWORD dwSize;
    DWORD dwAttackLevel;
    DWORD dwAttackTime;
    DWORD dwFadeLevel;
    DWORD dwFadeTime;
} DIENVELOPE, *LPDIENVELOPE;

typedef const DIENVELOPE *LPCDIENVELOPE;
```

Members

dwSize
The size, in bytes, of the structure. This member must be initialized before the structure is used.

dwAttackLevel
Amplitude for the start of the envelope, relative to the baseline, in the range 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

dwAttackTime
The time, in microseconds, to reach the sustain level.

dwFadeLevel
Amplitude for the end of the envelope, relative to the baseline, in the range 0 to 10,000. If the effect's type-specific data does not specify a baseline, then the amplitude is relative to zero.

dwFadeTime
The time, in microseconds, to reach the fade level.

DIJOYSTATE

The **DIJOYSTATE** structure contains information about the state of a joystick device. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct DIJOYSTATE {
    LONG    lX;
    LONG    lY;
    LONG    lZ;
    LONG    lRx;
    LONG    lRy;
    LONG    lRz;
    LONG    rglSlider[2];
    DWORD   rgdwPOV[4];
    BYTE    rgbButtons[32];
} DIJOYSTATE, *LPDIJOYSTATE;
```

Members

lX
Information about the joystick x-axis (usually the left-right movement of a stick).

lY
Information about the joystick y-axis (usually the forward-backward movement of a stick).

lZ
Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

lRx
Information about the joystick x-axis rotation. If the joystick does not have this, the value is zero.

lRy
Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is zero.

lRz
Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is zero.

rglSlider[2]
Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice::GetObjectInfo** method to obtain semantic information about these values.

rgdwPOV[4]
The current position of up to four direction controllers (such as point-of-view hats). The position is indicated in hundredths of degrees clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, *dwPOV* will be -1, 0, 9,000, 18,000, or 27,000.

rgbButtons[32]
Array of button states. The high-order bit of the byte is set if the corresponding button is down and clear if the button is up or does not exist.

Remarks

You must prepare the device for joystick-style access by calling the **IDirectInputDevice::SetDataFormat** method, passing the *c_dfDIJoystick* global data format variable.

If an axis is in relative mode, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the

indicator is centered as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

DIJOYSTATE2

The **DIJOYSTATE2** structure contains information about the state of a joystick device with extended capabilities. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct DIJOYSTATE2 {
    LONG    lX;
    LONG    lY;
    LONG    lZ;
    LONG    lRx;
    LONG    lRy;
    LONG    lRz;
    LONG    rglSlider[2];
    DWORD   rgdwPOV[4];
    BYTE    rgbButtons[128];
    LONG    lVX;
    LONG    lVY;
    LONG    lVZ;
    LONG    lVRx;
    LONG    lVRy;
    LONG    lVRz;
    LONG    rglVSlider[2];
    LONG    lAX;
    LONG    lAY;
    LONG    lAZ;
    LONG    lARx;
    LONG    lARy;
    LONG    lARz;
    LONG    rglASlider[2];
    LONG    lFX;
    LONG    lFY;
    LONG    lFZ;
    LONG    lFRx;
    LONG    lFRy;
    LONG    lFRz;
    LONG    rglFSlider[2];
} DIJOYSTATE2, *LPDIJOYSTATE2;
```

lX
Information about the joystick x-axis (usually the left-right movement of a stick).

lY
Information about the joystick y-axis (usually the forward-backward movement of a stick).

lZ
Information about the joystick z-axis (often the throttle control). If the joystick does not have this axis, the value is zero.

lRx
Information about the joystick x-axis rotation. If the joystick does not have this, the value is zero.

lRy
Information about the joystick y-axis rotation. If the joystick does not have this axis, the value is zero.

lRz
Information about the joystick z-axis rotation (often called the rudder). If the joystick does not have this axis, the value is zero.

rglSlider[2]

Two additional axis values (formerly called the u-axis and v-axis) whose semantics depend on the joystick. Use the **IDirectInputDevice::GetObjectInfo** method to obtain semantic information about these values.

rgdwPOV[4]
The current position of up to four direction controllers (such as point-of-view hats). The position is indicated in hundredths of degrees clockwise from north (away from the user). The center position is normally reported as -1; but see Remarks. For indicators that have only five positions, *dwPOV* will be -1, 0, 9,000, 18,000, or 27,000.

rgbButtons[128]
Array of button states. The high-order bit of the byte is set if the corresponding button is down and clear if the button is up or does not exist.

VX
Information about the x-axis velocity.

lVY
Information about the y-axis velocity.

lVZ
Information about the z-axis velocity.

lVRx
Information about the x-axis angular velocity.

lVRy
Information about the y-axis angular velocity.

lVRz
Information about the z-axis angular velocity.

rglVSlider[2]
Information about extra axis velocities.

lAX
Information about the x-axis acceleration.

lAY
Information about the y-axis acceleration.

lAZ
Information about the z-axis acceleration.

lARx
Information about the x-axis angular acceleration.

lARy
Information about the y-axis angular acceleration.

lARz
Information about the z-axis angular acceleration.

rglASlider[2]
Information about extra axis accelerations.

lFX
Information about the x-axis force.

lFY
Information about the y-axis force.

lFZ
Information about the z-axis force.

lFRx
Information about the x-axis torque.

lFRy
Information about the y-axis torque.

lFRz

Information about the z-axis torque.

rglFSlider[2]

Information about extra axis forces.

Remarks

You must prepare the device for access to a joystick with extended capabilities by calling the **IDirectInputDevice::SetDataFormat** method, passing the *c_dfDIJoystick2* global data format variable.

The **DIJOYSTATE2** structure has no special association with the **IDirectInputDevice2** interface. You can use either **DIJOYSTATE** or **DIJOYSTATE2** with either the **IDirectInputDevice** or the **IDirectInputDevice2** interface.

If an axis is in relative mode, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

Some drivers report the centered position of the POV indicator as 65,535. Determine whether the indicator is centered as follows:

```
BOOL POVCentered = (LOWORD(dwPOV) == 0xFFFF);
```

DIMOUSESTATE

The **DIMOUSESTATE** structure contains information about the state of a mouse device or another device that is being accessed as if it were a mouse device. This structure is used with the **IDirectInputDevice::GetDeviceState** method.

```
typedef struct {
    LONG lX;
    LONG lY;
    LONG lZ;
    BYTE rgbButtons[4];
} DIMOUSESTATE, *LPDIMOUSESTATE;
```

Members

lX
Information about the mouse x-axis.

lY
Information about the mouse y-axis.

lZ
Information about the mouse z-axis (typically a wheel). If the mouse does not have a z-axis, then the value is zero.

rgbButtons[4]
Array of button states. The high-order bit of the byte is set if the corresponding button is down.

Remarks

You must prepare the device for mouse-style access by calling the **IDirectInputDevice::SetDataFormat** method, passing the *c_dfDIMouse* global data format variable.

The mouse is a relative-axis device, so the absolute axis positions for mouse axes are simply accumulated relative motion. As a result, the value of the absolute axis position is not meaningful except in comparison with other absolute axis positions.

If an axis is in relative mode, then the appropriate member contains the change in position. If it is in absolute mode, then the member contains the absolute axis position.

DIOBJECTDATAFORMAT

The **DIOBJECTDATAFORMAT** structure contains information about a device object's data format for use with the **IDirectInputDevice::SetDataFormat** method.

```
typedef struct {
    const GUID * pguid;
    DWORD        dwOfs;
    DWORD        dwType;
    DWORD        dwFlags;
} DIOBJECTDATAFORMAT, *LPDIOBJECTDATAFORMAT;

typedef const DIOBJECTDATAFORMAT *LPCDIOBJECTDATAFORMAT;
```

Members

pguid

Unique identifier for the axis, button, or other input source. When requesting a data format, making this member NULL indicates that any type of object is permissible.

dwOfs

Offset within the data packet where the data for the input source will be stored. This value must be a multiple of four for **DWORD** size data, such as axes. It can be byte-aligned for buttons.

dwType

Device type that describes the object. It is a combination of the following flags describing the object type (axis, button, and so forth) and containing the object-instance number in the middle 16 bits. When requesting a data format, the instance portion must be set to DIDFT_ANYINSTANCE to indicate that any instance is permissible, or to DIDFT_MAKEINSTANCE(*n*) to restrict the request to instance *n*. See the examples under Remarks.

DIDFT_ABSAXIS

The object selected by the **IDirectInput::SetDataFormat** method must be an absolute axis.

DIDFT_AXIS

The object selected by the **IDirectInput::SetDataFormat** method must be an absolute or relative axis.

DIDFT_BUTTON

The object selected by the **IDirectInput::SetDataFormat** method must be a push button or a toggle button.

DIDFT_FFACTUATOR

The object selected by the **IDirectInput::SetDataFormat** method must contain a force feedback actuator; in other words, it must be possible to apply forces to the object.

DIDFT_FFEFFECTTRIGGER

The object selected by the **IDirectInput::SetDataFormat** method must be a valid force feedback effect trigger.

DIDFT_POV

The object selected by the **IDirectInput::SetDataFormat** method must be a point-of-view controller.

DIDFT_PSHBUTTON

The object selected by the **IDirectInput::SetDataFormat** method must be a push button.

DIDFT_RELAXIS

The object selected by **IDirectInputDevice::SetDataFormat** must be a relative axis.

DIDFT_TGLBUTTON

The object selected by **IDirectInputDevice::SetDataFormat** must be a toggle button.

dwFlags

Zero or more of the following values:

DIDOI_ASPECTACCEL

The object selected by **IDirectInputDevice::SetDataFormat** must report acceleration information.

DIDOI_ASPECTFORCE

The object selected by **IDirectInputDevice::SetDataFormat** must report force information.

DIDOI_ASPECTPOSITION

The object selected by **IDirectInputDevice::SetDataFormat** must report position information.

DIDOI_ASPECTVELOCITY

The object selected by **IDirectInputDevice::SetDataFormat** must report velocity information.

Remarks

A data format is made up of several **DIOBJECTDATAFORMAT** structures, one for each object (axis, button, and so on). An array of these structures is contained in the **DIDATAFORMAT** structure that is passed to **IDirectInputDevice::SetDataFormat**. An application typically does not need to create an array of **DIOBJECTDATAFORMAT** structures; rather, it can use one of the predefined data formats, *c_dfDIMouse*, *c_dfDIKeyboard*, *c_dfDIJoystick*, or *c_dfDIJoystick2*, which have predefined settings for **DIOBJECTDATAFORMAT**.

The following object data format specifies that DirectInput should choose the first available axis and report its value in the **DWORD** at offset 4 in the device data.

```
DIOBJECTDATAFORMAT dfAnyAxis = {
    0,                              // Wildcard
    4,                              // Offset
    DIDFT_AXIS | DIDFT_ANYINSTANCE, // Any axis is okay
    0,                              // Don't care about aspect
};
```

The following object data format specifies that the x-axis of the device should be stored in the **DWORD** at offset 12 in the device data. If the device has more than one x-axis, the first available one should be selected.

```
DIOBJECTDATAFORMAT dfAnyXAxis = {
    &GUID_XAxis,                    // Must be an X axis
    12,                             // Offset
    DIDFT_AXIS | DIDFT_ANYINSTANCE, // Any X axis is okay
    0,                              // Don't care about aspect
};
```

The following object data format specifies that DirectInput should choose the first available button and report its value in the high bit of the byte at offset 16 in the device data.

```
DIOBJECTDATAFORMAT dfAnyButton = {
    0,                                  // Wildcard
    16,                                 // Offset
    DIDFT_BUTTON | DIDFT_ANYINSTANCE, // Any button is okay
    0,                                  // Don't care about aspect
};
```

The following object data format specifies that button 0 of the device should be reported as the high bit of the byte stored at offset 18 in the device data.

If the device does not have a button 0, the attempt to set this data format will fail.

```
DIOBJECTDATAFORMAT dfButton0 = {
    0,                                      // Wildcard
    18,                                     // Offset
    DIDFT_BUTTON | DIDFT_MAKEINSTANCE(0), // Button zero
    0,                                      // Don't care about aspect
};
```

DIPERIODIC

The **DIPERIODIC** structure contains type-specific information for effects that are marked as **DIEFT_PERIODIC**.

The structure describes a periodic effect.

A pointer to a single **DIPERIODIC** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct {
    DWORD dwMagnitude;
    LONG lOffset;
    DWORD dwPhase;
    DWORD dwPeriod;
} DIPERIODIC, *LPDIPERIODIC;

typedef const DIPERIODIC *LPCDIPERIODIC;
```

Members

dwMagnitude
The magnitude of the effect, in the range 0 to 10,000. If an envelope is applied to this effect, then the value represents the magnitude of the sustain. If no envelope is applied, then the value represents the amplitude of the entire effect.

lOffset
The offset of the effect. The range of forces generated by the effect will be **lOffset** - **dwMagnitude** to **lOffset** + **dwMagnitude**. The value of the **lOffset** member is also the baseline for any envelope that is applied to the effect.

dwPhase
The position in the cycle of the periodic effect at which playback begins, in the range 0 to 35,999. See Remarks.

dwPeriod
The period of the effect in microseconds.

Remarks

A device driver may not provide support for all values in the **dwPhase** member. In this case the value will be rounded off to the nearest supported value.

DIPROPDWORD

The **DIPROPDWORD** is a generic structure used to access **DWORD** properties.

```
typedef struct {
    DIPROPHEADER     diph;
    DWORD            dwData;
} DIPROPDWORD, *LPDIPROPDWORD;

typedef const DIPROPDWORD *LPCDIPROPDWORD;
```

Members

diph

A **DIPROPHEADER** structure that must be initialized as follows:

Member	Value
dwSize	sizeof(**DIPROPDWORD**)
dwHeaderSize	sizeof(**DIPROPHEADER**)
dwObj	If the **dwHow** member is DIPH_DEVICE, this member must be zero. If the **dwHow** member is DIPH_BYID, this member must be the identifier for the object whose property setting is to be set or retrieved. If the **dwHow** member is DIPH_BYOFFSET, this member must be a data format offset for the object whose property setting is to be set or retrieved. For example, if the *c_dfDIMouse* data format is selected, it must be one of the DIIMOFS_* values.
dwHow	Specifies how the **dwObj** member should be interpreted. If **dwObj** is DIPROP_AXISMODE or DIPROP_BUFFERSIZE, **dwHow** should be DIPH_DEVICE.

dwData

The property-specific value being set or retrieved.

See Also

DIPROPRANGE, **IDirectInputDevice::GetProperty**, **IDirectInputDevice::SetProperty**

DIPROPHEADER

The **DIPROPHEADER** is a generic structure that is placed at the beginning of all property structures.

```
typedef struct {
    DWORD    dwSize;
    DWORD    dwHeaderSize;
    DWORD    dwObj;
    DWORD    dwHow;
} DIPROPHEADER, *LPDIPROPHEADER;

typedef const DIPROPHEADER *LPCDIPROPHEADER;
```

Members

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

dwHeaderSize
Size of the **DIPROPHEADER** structure.

dwObj
Object for which the property is to be accessed. The value set for this member depends on the value specified in the **dwHow** member.

dwHow
Value specifying how the **dwObj** member should be interpreted. This value can be one of the following:

Value	Meaning
DIPH_DEVICE	The **dwObj** member must be zero.
DIPH_BYOFFSET	The **dwObj** member is the offset into the current data format of the object whose property is being accessed.
DIPH_BYID	The **dwObj** member is the object type/instance identifier. This identifier is returned in the **dwType** member of the **DIDEVICEOBJECTINSTANCE** structure returned from a previous call to the **IDirectInputDevice::EnumObjects** member.

DIPROPRANGE

The **DIPROPRANGE** structure contains information about the range of an object within a device. This structure is used with the DIPROP_RANGE flag set in the **IDirectInputDevice::GetProperty** and **IDirectInputDevice::SetProperty** methods.

```
typedef struct {
    DIPROPHEADER diph;
    LONG         lMin;
    LONG         lMax;
} DIPROPRANGE, *LPDIPROPRANGE;

typedef const DIPROPRANGE *LPCDIPROPRANGE;
```

Members

diph

A **DIPROPHEADER** structure that must be initialized as follows:

dwSize	sizeof(DIPROPRANGE)
dwHeaderSize	sizeof(**DIPROPHEADER**)
dwObj	Identifier of the object whose range is being retrieved or set.
dwHow	How the **dwObj** member should be interpreted.

lMin

The lower limit of the range, inclusive. If the range of the device is unrestricted, this value will be DIPROPRANGE_NOMIN when the **IDirectInputDevice::GetProperty** method returns.

lMax

The upper limit of the range, inclusive. If the range of the device is unrestricted, this value will be DIPROPRANGE_NOMAX when the **IDirectInputDevice::GetProperty** method returns.

Remarks

The range values for devices whose ranges are unrestricted will wrap around.

See Also

DIPROPDWORD, **IDirectInputDevice::GetProperty**, **IDirectInputDevice::SetProperty**

DIRAMPFORCE

The **DIRAMPFORCE** structure contains type-specific information for effects that are marked as **DIEFT_RAMPFORCE**. The structure describes a ramp force effect.

A pointer to a single **DIRAMPFORCE** structure for an effect is passed in the **lpvTypeSpecificParams** member of the **DIEFFECT** structure.

```
typedef struct {
    LONG lStart;
    LONG lEnd;
} DIRAMPFORCE, *LPDIRAMPFORCE;

typedef const DIRAMPFORCE *LPCDIRAMPFORCE;
```

Members

lStart
The magnitude at the start of the effect, in the range -10,000 to +10,000.

lEnd
The magnitude at the end of the effect, in the range -10,000 to +10,000.

Remarks

The **dwDuration** for a ramp force effect cannot be INFINITE.

Device Constants

This section is a reference for constants used to interpret data for keys, buttons, and axes.

- Keyboard Device Constants
- DirectInput and Japanese Keyboards
- Mouse Device Constants
- Joystick Device Constants

Keyboard Device Constants

Keyboard device constants, defined in Dinput.h, represent offsets within a keyboard device's data packet, a 256-byte array. The data at a given offset is associated with a keyboard key. Typically, these values will be used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures, or as indices when accessing data within the array using array notation.

The standard keyboard device constants are the following (in ascending order):

Constant	Note
DIK_ESCAPE	
DIK_1	On main keyboard
DIK_2	On main keyboard
DIK_3	On main keyboard
DIK_4	On main keyboard
DIK_5	On main keyboard
DIK_6	On main keyboard
DIK_7	On main keyboard
DIK_8	On main keyboard
DIK_9	On main keyboard
DIK_0	On main keyboard
DIK_MINUS	On main keyboard
DIK_EQUALS	On main keyboard
DIK_BACK	The BACKSPACE key
DIK_TAB	
DIK_Q	
DIK_W	
DIK_E	
DIK_R	
DIK_T	
DIK_Y	
DIK_U	
DIK_I	
DIK_O	
DIK_P	
DIK_LBRACKET	The [key
DIK_RBRACKET	The] key
DIK_RETURN	ENTER key on main keyboard
DIK_LCONTROL	Left CTRL key
DIK_A	
DIK_S	
DIK_D	
DIK_F	
DIK_G	
DIK_H	
DIK_J	

DIK_K	
DIK_L	
DIK_SEMICOLON	
DIK_APOSTROPHE	
DIK_GRAVE	Grave accent (`) key
DIK_LSHIFT	Left SHIFT key
DIK_BACKSLASH	
DIK_Z	
DIK_X	
DIK_C	
DIK_V	
DIK_B	
DIK_N	
DIK_M	
DIK_COMMA	
DIK_PERIOD	On main keyboard
DIK_SLASH	Forward slash on main keyboard
DIK_RSHIFT	Right SHIFT key
DIK_MULTIPLY	The * key on numeric keypad
DIK_LMENU	Left ALT key
DIK_SPACE	SPACEBAR
DIK_CAPITAL	CAPS LOCK key
DIK_F1	
DIK_F2	
DIK_F3	
DIK_F4	
DIK_F5	
DIK_F6	
DIK_F7	
DIK_F8	
DIK_F9	
DIK_F10	
DIK_NUMLOCK	
DIK_SCROLL	SCROLL LOCK
DIK_NUMPAD7	
DIK_NUMPAD8	
DIK_NUMPAD9	
DIK_SUBTRACT	MINUS SIGN on numeric keypad
DIK_NUMPAD4	
DIK_NUMPAD5	
DIK_NUMPAD6	
DIK_ADD	PLUS SIGN on numeric keypad
DIK_NUMPAD1	
DIK_NUMPAD2	

DIK_NUMPAD3	
DIK_NUMPAD0	
DIK_DECIMAL	PERIOD (decimal point) on numeric keypad
DIK_F11	
DIK_F12	
DIK_F13	
DIK_F14	
DIK_F15	
DIK_KANA	On Japanese keyboard
DIK_CONVERT	On Japanese keyboard
DIK_NOCONVERT	On Japanese keyboard
DIK_YEN	On Japanese keyboard
DIK_NUMPADEQUALS	On numeric keypad (NEC PC98)
DIK_CIRCUMFLEX	On Japanese keyboard
DIK_AT	On Japanese keyboard
DIK_COLON	On Japanese keyboard
DIK_UNDERLINE	On Japanese keyboard
DIK_KANJI	On Japanese keyboard
DIK_STOP	On Japanese keyboard
DIK_AX	On Japanese keyboard
DIK_UNLABELED	On Japanese keyboard
DIK_NUMPADENTER	
DIK_RCONTROL	Right CTRL key
DIK_NUMPADCOMMA	COMMA on NEC PC98 numeric keypad
DIK_DIVIDE	Forward slash on numeric keypad
DIK_SYSRQ	
DIK_RMENU	Right ALT key
DIK_HOME	
DIK_UP	UP ARROW
DIK_PRIOR	PAGE UP
DIK_LEFT	LEFT ARROW
DIK_RIGHT	RIGHT ARROW
DIK_END	
DIK_DOWN	DOWN ARROW
DIK_NEXT	PAGE DOWN
DIK_INSERT	
DIK_DELETE	
DIK_LWIN	Left Windows key
DIK_RWIN	Right Windows key
DIK_APPS	Application key

The following alternate names are available:

Alternate name	**Regular name**	**Note**
DIK_BACKSPACE	DIK_BACK	BACKSPACE
DIK_NUMPADSTAR	DIK_MULTIPLY	* key on numeric

		keypad
DIK_LALT	DIK_LMENU	Left ALT
DIK_CAPSLOCK	DIK_CAPITAL	CAPSLOCK
DIK_NUMPADMINUS	DIK__SUBTRACT	Minus key on numeric keypad
DIK_NUMPADPLUS	DIK_ADD	Plus key on numeric keypad
DIK_NUMPADPERIOD	DIK_DECIMAL	Period key on numeric keypad
DIK_NUMPADSLASH	DIK__DIVIDE	Forward slash on numeric keypad
DIK_RALT	DIK_RMENU	Right ALT
DIK_UPARROW	DIK_UP	On arrow keypad
DIK_PGUP	DIK_PRIOR	On arrow keypad
DIK_LEFTARROW	DIK_LEFT	On arrow keypad
DIK_RIGHTARROW	DIK_RIGHT	On arrow keypad
DIK_DOWNARROW	DIK_DOWN	On arrow keypad
DIK_PGDN	DIK_NEXT	On arrow keypad

For information on Japanese keyboards, see DirectInput and Japanese Keyboards.

DirectInput and Japanese Keyboards

There are substantial differences between Japanese and U.S. keyboards. The chart below lists the additional keys that are available on each type of Japanese keyboard. It also lists the keys that are available on U.S. keyboards but are missing on the various Japanese keyboards.

Also note that on some NEC PC-98 keyboards, the DIK_CAPSLOCK and DIK_KANA keys are toggle buttons and not push buttons. These generate a down event when first pressed, then generate an up event when pressed a second time.

Keyboard	Additional Keys	Missing Keys
DOS/V 106 Keyboard, NEC PC-98 106 Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_CONVERT, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_APOSTROPHE, DIK_EQUALS, DIK_GRAVE
NEC PC-98 Standard Keyboard, NEC PC-98 Laptop Keyboard	DIK_AT, DIK_CIRCUMFLEX, DIK_COLON, DIK_F13, DIK_F14, DIK_F15, DIK_KANA, DIK_KANJI, DIK_NOCONVERT, DIK_NUMPADCOMMA, DIK_NUMPADEQUALS , DIK_STOP, DIK_UNDERLINE, DIK_YEN	DIK_APOSTROPHE, DIK_BACKSLASH, DIK_EQUALS, DIK_GRAVE , DIK_NUMLOCK, DIK_NUMPADENTER, DIK_RCONTROL, DIK_RMENU, DIK_RSHIFT , DIK_SCROLL
AX Keyboard	DIK_AX, DIK_CONVERT, DIK_KANJI, DIK_NOCONVERT, DIK_YEN	DIK_RCONTROL, DIK_RMENU
J-3100 Keyboard	DIK_KANA, DIK_KANJI, DIK_NOLABEL, DIK_YEN	DIK_RCONTROL, DIK_RMENU

Mouse Device Constants

Mouse device constants, defined in Dinput.h, represent offsets within a mouse device's data packet, the **DIMOUSESTATE** structure. The data at a given offset is associated with a device object (button or axis). Typically, these values will be used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures.

The mouse device constants are the following:

DIMOFS_BUTTON0	Offset of the data representing the state of mouse button 0.
DIMOFS_BUTTON1	Offset of the data representing the state of mouse button 1.
DIMOFS_BUTTON2	Offset of the data representing the state of mouse button 2.
DIMOFS_BUTTON3	Offset of the data representing the state of mouse button 3.
DIMOFS_X	Offset of the data representing the mouse's position on the x-axis.
DIMOFS_Y	Offset of the data representing the mouse's position on the y-axis.
DIMOFS_Z	Offset of the data representing the mouse's position on the z-axis.

Joystick Device Constants

Joystick device constants represent offsets within a joystick device's data packet, the **DIJOYSTATE** structure. The data at a given offset is associated with a device object; that is, a button or axis. Typically, these values will be used in the **dwOfs** member of the **DIDEVICEOBJECTDATA**, **DIOBJECTDATAFORMAT** or **DIDEVICEOBJECTINSTANCE** structures.

The following macros return a constant indicating the offset of the data for a particular button or axis relative to the beginning of the **DIJOYSTATE** structure:

DIJOFS_BUTTON0 *to* DIJOFS_BUTTON31 *or* DIJOFS_BUTTON(*n*)	A button.
DIJOFS_POV(*n*)	A point-of-view indicator.
DIJOFS_RX	The x-axis rotation.
DIJOFS_RY	The y-axis rotation.
DIJOFS_RZ	The z-axis rotation (rudder).
DIJOFS_X	The x-axis.
DIJOFS_Y	The y-axis.
DIJOFS_Z	The z-axis.
DIJOFS_SLIDER(*n*)	A slider axis.

Return Values

This table lists the **HRESULT** values that can be returned by DirectInput methods and functions. Errors are represented by negative values and cannot be combined.

For a list of the error values each method or function can return, see the individual descriptions. Lists of error codes in the documentation are necessarily incomplete. For example, any DirectInput method can return DIERR_OUTOFMEMORY even though the error code is not explicitly listed as a possible return value in the documentation for that method.

DI_BUFFEROVERFLOW

The device buffer overflowed and some input was lost. This value is equal to the S_FALSE standard COM return value.

DI_DOWNLOADSKIPPED

The parameters of the effect were successfully updated, but the effect could not be downloaded because the associated device was not acquired in exclusive mode.

DI_EFFECTRESTARTED

The effect was stopped, the parameters were updated, and the effect was restarted.

DI_NOEFFECT

The operation had no effect. This value is equal to the S_FALSE standard COM return value.

DI_NOTATTACHED

The device exists but is not currently attached. This value is equal to the S_FALSE standard COM return value.

DI_OK

The operation completed successfully. This value is equal to the S_OK standard COM return value.

DI_POLLEDDEVICE

The device is a polled device. As a result, device buffering will not collect any data and event notifications will not be signaled until the **IDirectInputDevice2::Poll** method is called.

DI_PROPNOEFFECT

The change in device properties had no effect. This value is equal to the S_FALSE standard COM return value.

DI_TRUNCATED

The parameters of the effect were successfully updated, but some of them were beyond the capabilities of the device and were truncated to the nearest supported value.

DI_TRUNCATEDANDRESTARTED

Equal to DI_EFFECTRESTARTED | DI_TRUNCATED.

DIERR_ACQUIRED

The operation cannot be performed while the device is acquired.

DIERR_ALREADYINITIALIZED

This object is already initialized

DIERR_BADDRIVERVER

The object could not be created due to an incompatible driver version or mismatched or incomplete driver components.

DIERR_BETADIRECTINPUTVERSION

The application was written for an unsupported prerelease version of DirectInput.

DIERR_DEVICEFULL

The device is full.

DIERR_DEVICENOTREG

The device or device instance is not registered with DirectInput. This value is equal to the REGDB_E_CLASSNOTREG standard COM return value.

DIERR_EFFECTPLAYING

The parameters were updated in memory but were not downloaded to the device because the device does not support updating an effect while it is still playing.

DIERR_HASEFFECTS

The device cannot be reinitialized because there are still effects attached to it.

DIERR_GENERIC

An undetermined error occurred inside the DirectInput subsystem. This value is equal to the E_FAIL standard COM return value.

DIERR_HANDLEEXISTS

The device already has an event notification associated with it. This value is equal to the E_ACCESSDENIED standard COM return value.

DIERR_INCOMPLETEEFFECT

The effect could not be downloaded because essential information is missing. For example, no axes have been associated with the effect, or no type-specific information has been supplied.

DIERR_INPUTLOST

Access to the input device has been lost. It must be reacquired.

DIERR_INVALIDPARAM

An invalid parameter was passed to the returning function, or the object was not in a state that permitted the function to be called. This value is equal to the E_INVALIDARG standard COM return value.

DIERR_MOREDATA

Not all the requested information fitted into the buffer.

DIERR_NOAGGREGATION

This object does not support aggregation.

DIERR_NOINTERFACE

The specified interface is not supported by the object. This value is equal to the E_NOINTERFACE standard COM return value.

DIERR_NOTACQUIRED

The operation cannot be performed unless the device is acquired.

DIERR_NOTBUFFERED

The device is not buffered. Set the DIPROP_BUFFERSIZE property to enable buffering.

DIERR_NOTDOWNLOADED

The effect is not downloaded.

DIERR_NOTEXCLUSIVEACQUIRED

The operation cannot be performed unless the device is acquired in DISCL_EXCLUSIVE mode.

DIERR_NOTFOUND

The requested object does not exist.

DIERR_NOTINITIALIZED

This object has not been initialized.

DIERR_OBJECTNOTFOUND

The requested object does not exist.

DIERR_OLDDIRECTINPUTVERSION

The application requires a newer version of DirectInput.

DIERR_OTHERAPPHASPRIO

Another application has a higher priority level, preventing this call from succeeding. This value is equal to the E_ACCESSDENIED standard COM return value. This error can be returned when an application has only foreground access to a device but is attempting to acquire the device while in the background.

DIERR_OUTOFMEMORY

The DirectInput subsystem couldn't allocate sufficient memory to complete the call. This value is equal to the E_OUTOFMEMORY standard COM return value.

DIERR_READONLY

The specified property cannot be changed. This value is equal to the E_ACCESSDENIED standard COM return value.

DIERR_UNSUPPORTED

The function called is not supported at this time. This value is equal to the E_NOTIMPL standard COM return value.

E_PENDING

Data is not yet available.

DirectSetup

This section provides information about the DirectSetup component of the DirectX® Programmer's Reference in the Platform Software Development Kit (SDK). Information is divided into the following groups:

- About DirectSetup
- DirectSetup Overview
- DirectSetup Reference

About DirectSetup

DirectSetup is a simple application programming interface (API) that provides you with a one-call installation for the DirectX components. This is more than merely a convenience; DirectX is a complex product, and its installation is an involved task. You should not attempt to manually install DirectX.

In addition, DirectSetup provides an automated way to install the appropriate Microsoft® Windows® registry information for applications that use the DirectPlayLobby object. This registry information is required for the DirectPlayLobby object to enumerate and start the application.

DirectSetup includes the following API functions: **DirectXRegisterApplication**, **DirectXUnRegisterApplication**, **DirectXSetup**, **DirectXSetupSetCallback**, and **DirectXSetupCallbackFunction**. The functions **DirectXRegisterApplication**, **DirectXUnRegisterApplication**, **DirectXSetup**, and **DirectXSetupSetCallback** are provided by Microsoft. The function **DirectXSetupCallbackFunction** is an optional function supplied by applications that use DirectSetup.

DirectSetup Overview

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

- What's New In DirectSetup For DirectX 5?
- Using the DirectXSetup Function
- The Default Setup Process With DirectXSetup
- Customizing Setup With the DirectSetup Callback Function
- Preparing a DirectX Application for Installation
- Enabling AutoPlay

What's New In DirectSetup For DirectX 5?

DirectSetup now supports a callback function that provides notification of various types of events that occur during the setup of DirectX. This allows developers to customize the setup interface. For details, see Customizing Setup With the DirectSetup Callback Function.

Also new in this version of DirectSetup is the ability of DirectPlayLobby applications to remove registration information. For details, see **DirectXUnRegisterApplication**.

Using the DirectXSetup Function

Applications and games that depend on DirectX use the **DirectXSetup** function to install their system components into an existing Windows installation. It optionally updates the display and audio drivers to support DirectX during the DirectX installation process. This process is designed to happen smoothly, without adversely affecting the user's system. Older drivers are upgraded whenever possible to prevent reduced performance or stability of all DirectX applications on a computer.

DirectXSetup is provided to each application from Dsetup.dll, DSetup16.dll, and Dsetup32.dll. Therefore all three of these files are included with your product's setup program. You can find the declarations for DirectSetup in Dsetup.h.

Note The **DirectXSetup** function overwrites system components from previous versions of DirectX. For example, if you install DirectX 5 on a system that already has DirectX 3 components, all DirectX 3 components will be overwritten. Because all DirectX components comply with Component Object Model (COM) backward compatibility rules, software written for DirectX 3 will continue to function properly.

DirectX 5 requires the installation of all components. Previous versions of DirectX allowed the installation of individual DirectX components. However, the amount of disk space saved by this was minimal. Current DirectX components are tightly integrated together for maximum performance. Hence, they all need to be installed for any one of them to work.

The DirectX Programmer's Reference of the Platform SDK contains the \Redist directory. Setup programs that use the **DirectXSetup** function must distribute the appropriate files from this directory as specified in the End User License Agreement (EULA).

The Default Setup Process With DirectXSetup

The **DirectXSetup** function can tell when DirectX components, display drivers, and audio drivers need to be upgraded. It can also distinguish whether or not these components can be upgraded without adversely affecting the Windows operating system. This is said to be a “safe” upgrade. It is important to note that the upgrade is safe for the operating system, not necessarily for the applications running on the computer. Some hardware-dependent applications can be negatively affected by an upgrade that is safe for Windows.

By default, the **DirectXSetup** function performs only safe upgrades. If the upgrade of a device driver may adversely affect the operation of Windows, the upgrade is not performed.

During the setup process, DirectSetup creates a backup copy of the system components and drivers that are replaced. These can typically be restored in the event of an error.

When display or audio drivers are upgraded, the **DirectXSetup** function utilizes a database created by Microsoft to manage the process. The database contains information on existing drivers that are provided either by Microsoft, the manufacturers of the hardware, or the vendors of the hardware. This database describes the upgrade status of each driver, based on testing done at Microsoft and at other sites.

Customizing Setup With the DirectSetup Callback Function

DirectSetup for DirectX 5 allows developers to specify a setup callback function. In the DirectSetup documentation, the callback function is referred to as **DirectXSetupCallbackFunction**. However, the actual name of the callback function is supplied by the application setup program.

If it is provided, **DirectXSetupCallbackFunction** is called once for each DirectX component and device driver that can be upgraded by the **DirectXSetup** function. Note that the callback function is completely optional. It does not have to be provided.

If a callback function is not provided by the setup program, **DirectXSetup** will display status and error information by calling the **MessageBox** function. If a callback is provided, the information that would be used by **DirectXSetup** to display the status with **MessageBox** is passed as parameters to the **DirectXSetupCallbackFunction** callback function. The callback function can use this information to display a status or error message using **MessageBox**. It can also implement a custom user interface to display the status or error message.

Uses of the DirectSetup Callback Function

DirectXSetupCallbackFunction can be used to:

- *Display a user interface that is customized for the application*. A game, for example, could display the progress of DirectX installation as a flying saucer descending toward a planet. When setup is complete, the saucer could land and an amusing alien could disembark carrying a sign reading, “Success!”
- *Suppress the display of status and error messages.* Designers of programs that are for novice users may want to suppress error messages so that they can be handled by the setup program. This requires a larger-than-normal development effort for the setup program, but may be appropriate for the target audience.
- *Automatically handle status and error conditions.* Setup programs that suppress error messages should handle them automatically. If the users are not given error information, they should not be required to intervene when an error occurs.
- *Allow the user greater control when status and error messages occur.* It is often appropriate to allow knowledgeable users greater control than normal over the setup process. Caution should be taken, however, when using this approach. Allowing users greater-than-normal control over the setup process increases the chances that their system will be adversely affected during or after the setup.
- *Override the default behavior of the **DirectXSetup** function.* Although this is not recommended, it can be done. The user is typically notified when a setup program does this.

Providing a Callback Function to DirectSetup

If a setup program provides a callback function to DirectSetup, it does so by calling the **DirectXSetupSetCallback** function before the **DirectXSetup** function. A pointer to the callback function is passed as a parameter to the **DirectXSetupSetCallback** function. See **DirectXSetupSetCallback**, and An Example Callback Function for details.

Interpreting DirectSetup Flags in the Callback Function

When the callback function **DirectXSetupCallbackFunction** is called by the **DirectXSetup** function, it is passed a parameter that contains the reason that the callback function was invoked. If the reason is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the *pInfo* parameter points to a structure containing flags that summarize the **DirectXSetup** function's recommendations on how the upgrade of DirectX components, display drivers, and audio drivers should be performed. For a chart that summarizes the callback function flags, see **DirectXSetupCallbackFunction**. The structure member containing the flags is called **UpgradeFlags** .

The flags passed through the **UpgradeFlags** member of the structure that is pointed to by the *pInfo* parameter of the callback function are present when the *Reason* parameter of the callback function is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE. They occur in the following combinations:

Primary Upgrade Flags

These flags are mutually exclusive. One of them is always present in the **UpgradeFlags** structure member.

DSETUP_BC_UPGRADE_FORCE

DSETUP_BC_UPGRADE_KEEP

DSETUP_BC_UPGRADE_SAFE

DSETUP_BC_UPGRADE_UNKNOWN

Secondary Upgrade Flags

Any or all of these flags may be present in the **UpgradeFlags** structure member.

DSETUP_BC_UPGRADE_CANTBACKUP

DSETUP_BC_UPGRADE_HASWARNINGS

Device Active Flag

This flag is present in the **UpgradeFlags** structure member if the device whose driver is being upgraded is active. This flag may be present in combination with any of the others.

DSETUP_BC_UPGRADE_DEVICE_ACTIVE

Device Class Flags

These flags are mutually exclusive. One of them is always present in the **UpgradeFlags** structure member.

DSETUP_BC_UPGRADE_DISPLAY

DSETUP_BC_UPGRADE_MEDIA

Every time the *Reason* parameter has the value DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the **UpgradeFlags** member of the structure pointed to by *pInfo* contains one Primary Upgrade Flag, zero or more Secondary Upgrade Flags, zero or one Device Active Flag, and one Device Class Flag.

If the **UpgradeFlags** member is set to DSETUP_CB_UPGRADE_KEEP, the DirectX component or device driver can't be upgraded. Performing an upgrade would cause Windows to cease to function properly. The **DirectXSetup** function will not perform an upgrade on the component or driver.

A value of DSETUP_CB_UPGRADE_FORCE in the **UpgradeFlags** structure member means that the component or driver *must* be upgraded for Windows to function properly. The **DirectXSetup** function will upgrade the driver or component. It is possible that the upgrade may adversely affect some programs on the system. When the **DirectXSetup** function detects this condition, the **UpgradeFlags** member will be set to DSETUP_CB_UPGRADE_FORCE | DSETUP_CB_UPGRADE_HAS_WARNINGS, where the symbol | represents a bitwise **OR** operation.

When this occurs, the **DirectXSetup** function will perform the upgrade, but issue a warning to the user.

Components and drivers are considered safe for upgrade if they will not adversely affect the operation of Windows when they are installed. In this case, the **UpgradeFlags** member will be set to DSETUP_CB_UPGRADE_SAFE. It is possible that the upgrade can be safe for Windows, but still cause problems for programs installed on the system. When **DirectXSetup** detects this condition, the **UpgradeFlags** member will contain the value DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNINGS. If this occurs, the default action for the **DirectXSetup** function is to not perform the upgrade.

If no callback is provided, the **DirectXSetup** function calls the Win32® API function **MessageBox** to get input from the user if needed. Typically, however, the **DirectXSetup** function will perform the default action without notifying the user. The function **DirectXSetupCallbackFunction** is supposed to return the same values that **MessageBox** would return if it were used.

The **MessageBox** function displays a message and some buttons for user response. When it is called, flags are passed to it that indicate what buttons should be present and which is the default button. These same flags are passed to the function **DirectXSetupCallbackFunction** in the *MsgType* parameter. These flags are the same flags that can be passed to the **MessageBox** function through its *uType* parameter.

The callback function should return what **MessageBox** would return if it were used. For instance, a callback function can be called with the flags in the *MsgType* parameter set to MB_YESNO | MB_DEFBUTTON1, where the | symbol is a bitwise **OR** operation. If **MessageBox** were called with these flags, it would present the user with a dialog box containing the **Yes** and **No** buttons. The default button is the **Yes** button. The callback should do something that is functionally equivalent to that. In this example, the return value of **MessageBox** would be the ID of the button that the user selected, either IDYES or IDNO. The return value of the callback function should be whichever one of these two the user selects.

A more complete discussion of the flags and the appropriate return values is contained in the Platform SDK documentation for the **MessageBox** function.

The following code is a function that can be used by DirectSetup callback functions. It illustrates the process of determining the ID of the default button for any allowable set of input flags.

```
INT DefaultButton(DWORD MsgType)
{
    INT iDefaultButton=0;

    switch (MsgType & 0x0F0F)
    {
        case MB_OK          | MB_DEFBUTTON1:
        case MB_OKCANCEL    | MB_DEFBUTTON1:
            iDefaultButton = IDOK;
        break;

        case MB_OKCANCEL    | MB_DEFBUTTON2:
        case MB_RETRYCANCEL | MB_DEFBUTTON2:
        case MB_YESNOCANCEL | MB_DEFBUTTON3:
            iDefaultButton = IDCANCEL;
        break;

        case MB_ABORTRETRYIGNORE | MB_DEFBUTTON1:
            iDefaultButton = IDABORT;
        break;

        case MB_RETRYCANCEL      | MB_DEFBUTTON1:
```

```
        case MB_ABORTRETRYIGNORE | MB_DEFBUTTON2:
            iDefaultButton = IDRETRY;
        break;

        case MB_ABORTRETRYIGNORE | MB_DEFBUTTON3:
            iDefaultButton = IDIGNORE;
        break;

        case MB_YESNO       | MB_DEFBUTTON1:
        case MB_YESNOCANCEL | MB_DEFBUTTON1:
            iDefaultButton = IDYES;
        break;

        case MB_YESNO       | MB_DEFBUTTON2:
        case MB_YESNOCANCEL | MB_DEFBUTTON2:
            iDefaultButton = IDNO;
        break;
    }

    return iDefaultButton;
}
```

In this example, the function uses bitwise **OR** operations to determine what kind of dialog box the **MessageBox** function would display, and which button is the default. A callback function can use a similar method to determine what value it should return.

Overriding DirectSetup Flags in the Callback Function

The function **DirectXSetupCallbackFunction** can override some of the default behaviors of the **DirectXSetup** function through its return value. As an example, the default behavior for **DirectXSetup** is to not install a component if the *UpgradeType* member of the *pInfo* parameter of the function **DirectXSetupCallbackFunction** is set to DSETUP_CB_UPGRADE_SAFE | DSETUP_CB_UPGRADE_HAS_WARNINGS (where the | symbol indicates a bitwise **OR** operation). In this case, the *MsgType* parameter of the callback function is set to MB_YESNO | MB_DEFBUTTON2. If the callback function accepts the default, it will return IDNO. If it wants to override the default, the callback function should return IDYES. If it does override the default, the user will be notified by the **DirectXSetup** function.

An Example Callback Function

A simple setup program could contain a callback function along the lines of the following:

```
DWORD MySetupCallback(
    DWORD Reason,
    DWORD MsgType,
    char *szMessage,
    char *szName,
    DSETUP_CB_UPGRADEINFO *pUpgradeInfo)
{
    if (MsgType==0)        // ignore status messages
        return ID_OK;

    return MessageBox(MyhWnd, szMessage,
                      "My Application Name", MsgType);
}
```

This example ignores all status messages, but displays error messages by calling the **MessageBox** function.

The address of the callback function in the example above would be passed to DirectSetup prior to calling the **DirectXSetup** function. The following code gives an example of how this is done.

```
// Set the callback function.
DirectXSetupSetCallback(MySetupCallback);
// Start the setup of DirectX components and drivers.
if (SUCCESS(DirectXSetup(hWndParent, NULL,
                         DSETUP_DIRECTX))
    {
        // The installation succeeded.
    }
    else
    {
        /* Installation failed. Handle the error in MySetupCallback.
        Do any additional cleanup needed right here. */
    }
```

Preparing a DirectX Application for Installation

At some point during the development of your application, you will need to create a setup program that installs your application and the DirectX files on a user's system. This program will determine the amount of disk space required to install your application and copy the appropriate DirectX files to the user's computer. You also need to create a directory on your distribution medium in which you will place all the application's files and any additional DirectX components. The following topics describe these steps:

- Creating the Setup Program
- Testing the Setup Program

Creating the Setup Program

The DirectX Programmer's Reference in the Platform SDK includes an example setup program that you can use as a model for your application's setup program. The setup program is called Dinstall, and it is located in the \Dxsdk\Sdk\Samples\Setup directory. It installs a sample DirectX application called Rockem. It also demonstrates one way to configure the **DirectXSetup** function.

To adapt the Dinstall.c program to your needs

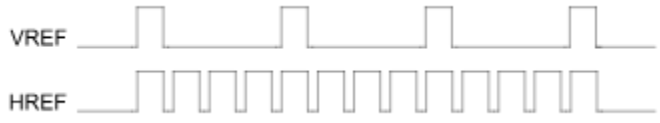
1. In an editor, open Dinstall.c.
2. Search for the text "copy_list".
3. Edit the list of files in this structure to contain the names of the files you want copied to the user's computer during installation.
4. If necessary, modify Dinstall.c so that it installs files in subdirectories on the user's hard disk. Currently, Dinstall installs files only in the default directory.
5. Search for two locations in Dinstall.c that contain the text "IDS_DISK_MSG".
6. Add some code that checks whether there is enough free hard disk space to install your application on the user's computer. Dinstall does not currently check this.

The *lpszRootPath* parameter of **DirectXSetup** specifies the path to the Dsetup*.dll files (Dsetup.dll, Dsetup16.dll, and Dsetup32.dll) and the Directx directory on your distribution media. These dynamic-link libraries and this directory should be located in the same directory as the Dinstall executable after it is compiled, unless there is an overwhelming reason to do otherwise. If all these files and directories are located in the same directory, the value of the *lpszRootPath* parameter should be set to NULL. This ensures that if the path changes when the files are placed on a compact disc or floppy disks from the root of the application, the **DirectXSetup** function still works properly.

For example, suppose Dinstall.exe, Dsetup*.dll, and the Directx directory are located in an application directory called D:\Funstuff during the testing phase. Then, when you burn the files on a compact disc, suppose you put them in the root. If the *lpszRootPath* parameter is set to "\FUNSTUFF", the setup program (Dinstall.exe) will not function from the compact disc. However, if the *lpszRootPath* parameter is set to NULL, the setup program will function in both cases, because the path to Dsetup*.dll, and the Directx directory are still in the current directory.

If you decide to place the Dsetup*.dll files and the Directx directory somewhere other than in the directory that contains Dinstall.exe, you must pass the appropriate parameters to **DirectXSetup** and load Dsetup.dll correctly. The *lpszRootPath* parameter of **DirectXSetup** should contain the full path to Dsetup.dll. In addition, you need to use the **LoadLibrary** and **GetProcAddress** Win32 functions in your setup program to locate Dsetup.dll.

The content of the **Setup** dialog box is determined by data supplied in the Dinstall.rc resource file.



To display your application's name and graphics

1. In an editor, open Dinstall.rc.
2. Search for all occurrences of "Rockem" and change them to the name of your application.
3. The graphics that are displayed in the **Setup** and **Reboot** dialog boxes are called Signon.bmp and Reboot.bmp in the resource file. Either rename your bitmap files these names, or change the names in the resource file to match the names of your bitmaps.
4. The icon for the Dinstall executable is called Setup.ico in the resource file, and it is specified by SETUP_ICON. Either set the name of your icon file to Setup.ico, or change the name in the

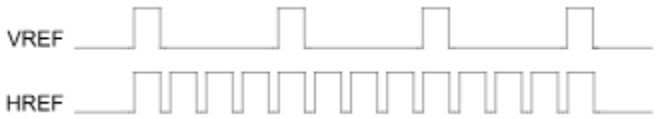
resource file to match the name of your icon file.

5. If appropriate for your application, change the default directory in which your application is installed. To do this, search for "IDS_DEFAULT_GAME_DIR" (it is located in two places in the resource file) and change the path of the default directory.

After you have modified the Dinstall.c and Dinstall.rc files to fit your application's needs, compile them into the Dinstall.exe executable file. You can rename this executable file to anything you want (Setup.exe, for example).

Testing the Setup Program

Before you commit your application to a compact disc or floppy disks, you should test your setup program. Do this by creating an application directory that contains all of your application's files, the setup program, and the DirectX files and drivers.



To set up the application directory

1 Create a directory that includes all your application's files. Be sure to create any subdirectories that are needed. Place the appropriate application files in the subdirectories.

2 Copy the setup program you wrote to the root of your application directory.

3 At the MS-DOS prompt, use the **xcopy** command to copy the Redist directory on the DirectX compact disc to the root of your application directory. For example, if your application's root directory is D:\Fungame, and the E: drive is your CD-ROM drive, type the following:

xcopy /s e:\redist*.* d:\fungame

Note The root of your application directory should include the entire contents of the Redist directory distributed on the DirectX Programmer's Reference on the Platform SDK. This is essential to ensure that the **DirectXSetup** function and the Dxsetup.exe file work properly.

Enabling AutoPlay

If you are building an AutoPlay compact disc title, copy the Autorun.inf file in the root directory of the DirectX Programmer’s Reference in the Platform SDK compact disc to the root of your application directory. This text file contains the following information:

```
[autorun]
OPEN=SETUP.EXE
```

If your application's setup program is called Setup.exe, you will not have to make any changes to this file; otherwise, edit this file to contain the name of your setup program. For more information, see **Autorun.inf**.

DirectSetup Reference

This section contains reference information for the API elements that DirectSetup provides. Reference material is divided into the following categories:

- Functions
- Structures
- Return Values

Functions

This section contains information on the following global functions used with DirectSetup:

- **DirectXRegisterApplication**
- **DirectXSetup**
- **DirectXSetupGetVersion**
- **DirectXSetupSetCallback**
- **DirectXSetupCallbackFunction**
- **DirectXUnRegisterApplication**

DirectXRegisterApplication

The **DirectXRegisterApplication** function registers an application as one designed to work with DirectPlayLobby.

```
int WINAPI DirectXRegisterApplication(
  HWND hWnd,
  LPDIRECTXREGISTERAPP lpDXRegApp
);
```

Parameters

hWnd

Handle to the parent window. If this parameter is set to NULL, the desktop is the parent window.

lpDXRegApp

Address of the **DIRECTXREGISTERAPP** structure that contains the registry entries that are required for the application to function properly with DirectPlayLobby.

Return Values

If this function is successful, it returns TRUE.

If it is not successful, it returns FALSE. Use the **GetLastError** Win32 function to get extended error information.

Remarks

The **DirectXRegisterApplication** function inserts the registry entries needed for an application to operate with DirectPlayLobby. If these registry entries are added with **DirectXRegisterApplication**, they should be removed with **DirectXUnRegisterApplication** when the application is uninstalled.

Many commercial install programs will remove registry entries automatically when a program in uninstalled. However, such a program will only do so if it added the registry entries itself. If the DirectPlayLobby registry entries are added by **DirectXRegisterApplication**, commercial install programs will not delete the registry entries when the application is uninstalled. Therefore, DirectPlayLobby registry entries that are created by **DirectXRegisterApplication** should be deleted by **DirectXUnRegisterApplication**.

Registry entries needed for DirectPlayLobby access can be created without the use of the **DirectXRegisterApplication** function. This, however, is not generally recommended. See *Registering Lobby-able Applications* in the DirectPlay® documentation.

See Also

DirectXUnRegisterApplication

DirectXSetup

The **DirectXSetup** function installs one or more DirectX components.

```
int WINAPI DirectXSetup(
  HWND hWnd,
  LPSTR lpszRootPath,
  DWORD dwFlags
);
```

Parameters

hWnd

Handle to the parent window for the setup dialog boxes.

lpszRootPath

Pointer to a string that contains the root path of the DirectX component files. This string must specify a full path to the directory that contains the files Dsetup.dll, Dsetup16.dll, and Dsetup.dll32. This directory is typically Redist. If you are certain the current directory contains Dsetup.dll and the Directx directory, this parameter can be NULL.

dwFlags

One or more flags indicating which DirectX components should be installed. A full installation (DSETUP_DIRECTX) is recommended.

DSETUP_D3D	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DDRAW	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DDRAWDRV	Installs display drivers provided by Microsoft.
DSETUP_DINPUT	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DIRECTX	Installs DirectX runtime components as well as DirectX-compatible display and audio drivers.
DSETUP_DIRECTXSETUP	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DPLAY	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DPLAYSP	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DSOUND	Obsolete. DirectX 3 programs that use this flag will install all DirectX components.
DSETUP_DSOUNDDRV	Installs audio drivers provided by Microsoft.
DSETUP_DXCORE	Installs DirectX runtime components. Does not install

	DirectX-compatible display and audio drivers.
DSETUP_TESTINSTALL	Performs a test installation. Does not actually install new components.

Return Values

If this function is successful, it returns SUCCESS.

If it is not successful, it returns an error code. For a list of possible return codes, see Return Values .

Remarks

Before you use the **DirectXSetup** function in your setup program, you should ensure that there is at least 15 MB of available disk space on the user's system. This is the maximum space required for DirectX to set up the appropriate files. If the user's system already contains the DirectX files, this space is not needed.

See Also

Using the DirectXSetup Function.

DirectXSetupGetVersion

The **DirectXSetupGetVersion** function retrieves the version number of the DirectX components that are currently installed.

```
INT WINAPI DirectXSetupGetVersion(
  DWORD   *pdwVersion  // receives the version number
  DWORD   *pdwRevision // receives the revision number
);
```

Parameters

pdwVersion

Pointer to a **DWORD**. The **DirectXSetupGetVersion** function will fill the **DWORD** with the version number. If this parameter is NULL, it is ignored.

pdwRevision

Pointer to a **DWORD**. The **DirectXSetupGetVersion** function will fill the **DWORD** with the revision number. If this parameter is NULL, it is ignored.

Return Values

If this function is successful, it returns non-zero.

If it is not successful, it returns zero.

Remarks

The **DirectXSetupGetVersion** function can be used to retrieve the version and revision numbers before or after the **DirectXSetup** function is called. If it is called before the **DirectXSetup** function is invoked, it gives the version and revision numbers of the DirectX components that are currently installed. If it is called after the **DirectXSetup** function is called, but before the computer has been rebooted, it will give the version and revision numbers of the DirectX components that will take effect after the computer is restarted.

The version number in the *pdwVersion* parameter is composed of the major version number and the minor version number. The major version number will be in the 16 most significant bits of the **DWORD** when this function returns. The minor version number will be in the 16 least significant bits of the **DWORD** when this function returns. The version numbers can be interpreted as follows:

DirectX Version	**Value Pointed At By *pdwVersion***
DirectX 1	0x00040001
DirectX 2	0x00040002
DirectX 3	0x00040003
DirectX 4	0x00040004
DirectX 5	0x00040005

The version number in the *pdwRevision* parameter is composed of the release number and the build number. The release number will be in the 16 most significant bits of the **DWORD** when this function returns. The build number will be in the 16 least significant bits of the **DWORD** when this function returns.

The following sample code fragment demonstrates how the information returned by **DirectXSetupGetVersion** can be extracted and used.

```
DWORD dwVersion;
DWORD dwRevision;
if (DirectXSetupGetVersion(&dwVersion, &dwRevision))
{
    printf("DirectX version is %d.%d.%d.%d\n",
```

```
            HIWORD(dwVersion), LOWORD(dwVersion),
            HIWORD(dwRevision), LOWORD(dwRevision));
}
```

Version and revision numbers can concatenated into a 64 bit quantity for comparison. The version number should be in the 32 most significant bits and the revision number should be in the 32 least significant bits.

See Also

DirectXSetup

DirectXSetupSetCallback

The **DirectXSetupSetCallback** sets a pointer to a callback function that is periodically called by **DirectXSetup**. The callback function can be used for setup progress notification and to implement a custom user interface for an application's setup program. For information on the callback function, see **DirectXSetupCallbackFunction**. If a setup program does not provide a callback function, the **DirectXSetupSetCallback** function should not be invoked.

```
INT WINAPI DirectXSetupSetCallback(
  DSETUP_CALLBACK   Callback  // pointer to the callback function
);
```

Parameters

Callback
Pointer to a callback function.

Return Values

Currently returns zero.

Remarks

To set a callback function, **DirectXSetupSetCallback** must be called before the **DirectXSetup** function is called.

The name of the callback function passed to **DirectXSetupSetCallback** is supplied by the setup program. However, it must match the prototype given in **DirectXSetupCallbackFunction**

See Also

DirectXSetupCallbackFunction, **DirectXSetup**

DirectXSetupCallbackFunction

DirectXSetupCallbackFunction is a placeholder name for a callback function supplied by the setup program. The callback function reports the status of the current installation process. It also can provide information for use by the **MessageBox** function.

```
DWORD DirectXSetupCallbackFunction(
  DWORD  Reason,   // reason for the callback
  DWORD  MsgType,  // same as MessageBox
  char *szMessage, // message string
  char *szName     // depends on Reason
  void *pInfo      // upgrade information
);
```

Parameters

Reason

Reason for the callback. It can be one of the following values.

DSETUP_CB_MSG_BEGIN_INSTALL

DirectXSetup is about to begin installing DirectX components and device drivers.

DSETUP_CB_MSG_BEGIN_INSTALL_DRIVERS

DirectXSetup is about to begin installing device drivers.

DSETUP_CB_MSG_BEGIN_INSTALL_RUNTIME

DirectXSetup is about to begin installing DirectX components.

DSETUP_CB_MSG_BEGIN_RESTORE_DRIVERS

DirectXSetup is about to begin restoring previous drivers.

DSETUP_CB_MSG_CANTINSTALL_BETA

A pre-release beta version of Windows 95 was detected. The DirectX component or device driver can't be installed.

DSETUP_CB_MSG_CANTINSTALL_NOTWIN32

The operating system detected is not Windows 95 or Windows NT®. DirectX is not compatible with Windows 3.*x*.

DSETUP_CB_MSG_CANTINSTALL_NT

The DirectX component or device driver can't be installed on versions of Windows NT prior to version 4.0.

DSETUP_CB_MSG_CANTINSTALL_UNKNOWNOS

The operating system is unknown. The DirectX component or device driver can't be installed.

DSETUP_CB_MSG_CANTINSTALL_WRONGLANGUAGE

The DirectX component or device driver is not localized to the language being used by Windows.

DSETUP_CB_MSG_CANTINSTALL_WRONGPLATFORM

The DirectX component or device driver is for another type of computer.

DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE

Driver is being considered for upgrade. Verification from user is recommended.

DSETUP_CB_MSG_INTERNAL_ERROR

An internal error has occurred. Setup of the DirectX component or device driver has failed.

DSETUP_CB_MSG_NOMESSAGE

No message to be displayed. The callback function should return.

DSETUP_CB_MSG_NOTPREINSTALLEDONNT

The DirectX component or device driver can't be installed on the version of Windows NT in use.

DSETUP_CB_MSG_PREINSTALL_NT

DirectX is already installed on the version of Windows NT in use.

DSETUP_CB_MSG_SETUP_INIT_FAILED

Setup of the DirectX component or device driver has failed.

MsgType

Contains flags that control the display of a message box. These flags can be passed to the **MessageBox** function. An exception is when *MsgType* is equal to zero. In that case, the setup program can display status information but should not wait for input from the user.

szMessage

A localized character string containing error or status messages that can be displayed in a message box created with the **MessageBox** function.

szName

The value of *szName* is NULL unless the *Reason* parameter is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE. In that case, *szName* contains the name of driver to be upgraded.

pInfo

Pointer to a structure containing upgrade information. When *Reason* is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE, the setup program is in the process of upgrading a driver and asking the user whether the upgrade should take place. This structure contains information about the upgrade in its **UpgradeType** member, which can have the following values.

DSETUP_CB_UPGRADE_CANTBACKUP

The old system components can't be backed up. Upgrade can be performed, but the components or drivers can't be restored later.

DSETUP_CB_UPGRADE_DEVICE_ACTIVE

The device is currently in use.

DSETUP_CB_UPGRADE_DEVICE_DISPLAY

The device driver being upgraded is for a display device.

DSETUP_CB_UPGRADE_DEVICE_MEDIA

The device driver being upgraded is for a media device.

DSETUP_CB_UPGRADE_FORCE

Windows may not function correctly if the component is not upgraded. The upgrade will be performed.

DSETUP_CB_UPGRADE_HASWARNINGS

DirectXSetup can upgrade the driver for this device, but doing so may affect one or more programs on the system. *szMessage* contains the names of which programs may be affected. Upgrade not recommended.

DSETUP_CB_UPGRADE_KEEP

The system may fail if this device driver is upgraded. Upgrade not allowed.

DSETUP_CB_UPGRADE_SAFE

DirectXSetup can safely upgrade this device driver. Upgrade recommended. A safe upgrade will not adversely affect the operation of Windows. Some hardware-dependent programs may be adversely affected.

DSETUP_CB_UPGRADE_UNKNOWN

DirectXSetup does not recognize the existing driver for this device. This value will occur frequently. Upgrading may adversely affect the use of the device. It is strongly recommended that the upgrade not be performed.

DSETUP_CB_UPGRADE_UNNECESSARY

The existing device driver is newer than the driver being installed. An upgrade is not recommended.

Return Values

The return value should be the same as the **MessageBox** function, with one exception. If this function returns zero, the **DirectXSetup** function will act as if no callback function was present. That is, it will perform the default action for upgrade of the DirectX component or driver.

Remarks

The name of the **DirectXSetupCallbackFunction** is supplied by the setup program. The **DirectXSetupSetCallback** function is used to pass the address of the callback function to DirectSetup.

If *MsgType* is equal to zero, the setup program may display status information, but it should not wait for user input. This is useful for displaying ongoing status information.

See Also

MessageBox, **DirectXSetupSetCallback**, Customizing Setup With the DirectSetup Callback Function

DirectXUnRegisterApplication

The **DirectXUnRegisterApplication** function deletes the registration of an application designed to work with DirectPlayLobby.

```
int WINAPI DirectXUnRegisterApplication(
  HWND hWnd,
  LPGUID lpGUID
);
```

Parameters

hWnd

Handle to the parent window. Set this to NULL if the desktop is the parent window.

lpGUID

Pointer to a GUID that represents the DirectPlay application to be unregistered.

Return Values

If the function succeeds, the return value is TRUE meaning that the registration is successfully deleted.

If the function fails, the return value is FALSE.

Remarks

The **DirectXUnRegisterApplication** function removes registry the entries needed for an application to work with DirectPlayLobby. An uninstall program should only use **DirectXUnRegisterApplication** if it used **DirectXRegisterApplication** when the application was installed.

See Also

DirectXRegisterApplication

Structures

This section contains information about the following structures used with DirectSetup:

- **DIRECTXREGISTERAPP**
- **DSETUP_CB_UPGRADEINFO**

DIRECTXREGISTERAPP

The **DIRECTXREGISTERAPP** structure contains the registry entries needed for applications designed to work with DirectPlayLobby.

```
typedef struct _DIRECTXREGISTERAPP {
    DWORD  dwSize;
    DWORD  dwFlags;
    LPSTR  lpszApplicationName;
    LPGUID lpGUID;
    LPSTR  lpszFilename;
    LPSTR  lpszCommandLine;
    LPSTR  lpszPath;
    LPSTR  lpszCurrentDirectory;
} DIRECTXREGISTERAPP, *PDIRECTXREGISTERAPP, *LPDIRECTXREGISTERAPP;
```

Members

dwSize
Size of the structure. Must be initialized to the size of the **DIRECTXREGISTERAPP** structure.

dwFlags
Reserved for future use.

lpszApplicationName
Name of the application.

lpGUID
Globally unique identifier (GUID) of the application.

lpszFilename
Name of the executable file to be called.

lpszCommandLine
Command-line arguments for the executable file.

lpszPath
Path of the executable file.

lpszCurrentDirectory
Current directory. This is typically the same as **lpszPath**.

DSETUP_CB_UPGRADEINFO

The **DSETUP_CB_UPGRADEINFO** structure is passed as a parameter to the **DirectXSetupCallbackFunction**. It only contains valid information when the *Reason* parameter is DSETUP_CB_MSG_CHECK_DRIVER_UPGRADE. Callback functions can use it to get status information on the upgrade that is about to be done.

```
typedef struct _DSETUP_CB_UPGRADEINFO
{
    DWORD UpgradeFlags;
} DSETUP_CB_UPGRADEINFO;
```

Members

UpgradeFlags

A flag indicating the status of the upgrade. See the *pInfo* parameter of the **DirectXSetupCallbackFunction** function for details.

See Also

DirectXSetupCallbackFunction

Return Values

The **DirectXSetup** function can return the values listed blow. It can also return a standard COM error.

DSETUPERR_SUCCESS

Setup was successful and no restart is required.

DSETUPERR_SUCCESS_RESTART

Setup was successful and a restart 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 were incorrect.

DSETUPERR_BADWINDOWSVERSION

DirectX does not support the Windows version on the system.

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.

DSETUPERR_NOTPREINSTALLEDONNT

The version of Windows NT on the system does not contain the current version of DirectX. An older version of DirectX may be present, or DirectX may be absent altogether.

DSETUPERR_OUTOFDISKSPACE

The setup program ran out of disk space during installation.

DSETUPERR_SOURCEFILENOTFOUND

One of the required source files could not be found.

DSETUPERR_UNKNOWNOS

The operating system on your system is not currently supported.

DSETUPERR_USERHITCANCEL

The **Cancel** button was pressed before the application was fully installed.

AutoPlay

This section provides information about the AutoPlay feature. Information is divided into the following groups:

- About AutoPlay
- AutoPlay Overview
- AutoPlay Reference

About AutoPlay

Microsoft® AutoPlay is a feature of the Microsoft Windows® operating system. AutoPlay automates the procedures for installing and configuring products designed for Windows-based platforms that are distributed on compact discs. When you insert a disc containing AutoPlay into a CD-ROM drive on a computer running Windows, AutoPlay automatically starts an application on the disc that installs, configures, and runs the selected product.

You can use AutoPlay to install and run CD-ROM applications that run in Windows, whether the application is based on the MS-DOS® operating system, Windows 3.0, Windows 3.1, Windows 95, or Windows NT®. If you want your CD-ROM product to display the Microsoft Windows 95 logo, it must be enabled for AutoPlay.

Note MS-DOS, Windows versions prior to Windows 95, and Windows NT versions 3.51 and earlier do not support AutoPlay. Adding AutoPlay to a compact disc, however, does not hinder or alter user interaction on computers running one of these operating systems.

AutoPlay Overview

This topic contains general information about the AutoPlay component. The following topics are discussed:

- How AutoPlay Works
- Autorun.inf
- Tips for Writing AutoPlay Applications
- Suppressing AutoPlay
- AutoPlay for MS-DOS-Based Applications

How AutoPlay Works

The implementation of AutoPlay relies on the following items:

- *A set of 32-bit CD-ROM device drivers for Windows 95 and Windows NT*. These device drivers detect when a user inserts a compact disc into a CD-ROM drive; device drivers for MS-DOS or previous versions of Windows do not.
- *An Autorun.inf file on the compact disc.* When you insert a disc into a CD-ROM drive on a computer running Windows 95 or Windows NT, the system immediately checks to see if the disc has a personal computer file system. If it does, the system searches for a file named Autorun.inf. This file specifies the application that AutoPlay runs. It can contain other information as well. or more information, see Autorun.inf.
- *A startup application on the compact disc.* Although you can start any application on the disc 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 your product.

Autorun.inf

The Autorun.inf file is a text file located in the root directory of the CD on which your DirectX® application is shipped. This file contains the name of the startup program on the disc. This startup program runs automatically when the disc is inserted in the CD-ROM drive. The Autorun.inf file also contains the name of the file of the icon that you want to represent your application's CD in the Windows user interface. In addition, the Autorun.inf file can contain optional menu commands that you want added to the shortcut menu. These menu commands are displayed when the user right-clicks the CD-ROM icon.

The following is an example of a minimal Autorun.inf file.

```
[autorun]
open=filename.exe
icon=filename.ico
```

The **[autorun]** section identifies the lines that follow it as AutoPlay commands. An **[autorun]** section is required in every Autorun.inf file. The **open** command specifies the path and file name of the startup application. The **icon** command specifies the file name that contains the icon.

The Autorun.inf file also can contain architecture-specific sections for Windows NT 4.0 running on RISC processors. For each type of processor architecture, add a section to the Autorun.inf file that contains the file name of the startup application you want to run for that architecture. The following table lists the commands used for the architectures that AutoPlay supports.

Architecture	Section Title
368 or higher	[autorun]
MIPS	[autorun.mips]
DEC Alpha	[autorun.alpha]
PowerPC	[autorun.ppc]

The following example shows how to create an Autorun.inf file that runs different startup applications depending on the computer architecture:

```
[autorun]
open=filename.exe
icon=filename.ico

[autorun.mips]
open=filenam2.exe
icon=filename.ico

[autorun.alpha]
open=filenam3.exe
icon=filename.ico

[autorun.ppc]
open=filenam4.exe
icon=filename.ico
```

The shell checks for an architecture-specific section first. If it does not find one, it uses the information in the **[autorun]** section. After the shell finds a section, it ignores all the other sections, so each section must contain all the information for that architecture.

Tips for Writing AutoPlay Applications

This section presents helpful guidelines for writing AutoPlay applications:

Opening a Startup Application

Users should receive feedback soon after they insert an AutoPlay compact disc into the disc drive. Therefore, your startup application should be a small program that loads quickly. The startup application should clearly identify the title it plays and provide an easy way to cancel the operation.

Loading in the Background

Typically, the startup application presents users with a dialog box asking them if they would like to proceed. The user normally clicks an **OK** button to continue. Take advantage of the time the user spends reading the dialog box by starting another thread that begins loading the setup application. If the user clicks **OK**, your setup program will already be loading. This significantly reduces the user's perception of the amount of time it takes to load your application.

Conserving Hard Disk Space

Hard disk space is a limited resource. Here are a few hints for minimizing hard disk usage:

- *Don't put files on the user's hard drive.* Run your application from the compact disc directly, without running any installation application.
- *Keep installed files to a minimum.* If your application needs to use the hard disk, install only the functional components necessary to run the application. In addition, provide a way to uninstall these components from the hard disk. For more information about uninstalling an application, see the documentation included with the Microsoft Platform Software Development Kit (SDK).
- *Provide the user with caching control.* If your application needs to use the drive as a data cache, provide the user with options in the startup application that will discard the cached data when the user quits the title or game.

Using the Registry

The registry is a feature of Windows that supersedes the initialization (.ini) and application configuration files. For information about application programming interfaces that manipulate the registry, see the documentation included with the Platform SDK.

If your product 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 there are no registry entries for your product—which means your product is being used for the first time—you could display a dialog box that lists the setup options. If your product is listed in the registry—which means it has already been installed—you could skip the setup options.

By changing the system registry, you can cause a computer to read the Autorun.inf file from a floppy disk. This feature of implementing AutoPlay on a floppy disk is provided only to help you debug your Autorun.inf files before you burn the compact disc. AutoPlay is intended for public distribution on compact disc only. To implement AutoPlay on a floppy disk, carry out the following procedure:

1. In Registry Editor (Regedit.exe), click **Edit**, and then click **Find**.
2. In the **Find What** box, type the following, and then click **Find Next**:

   NoDriveTypeAutoRun
3. Click **Edit**, and then click **Modify**.
4. Change the data of the NoDriveTypeAutoRun value from 0000 95 00 00 00 to 0000 91 00 00 00, and then click **OK**.

   This enables AutoPlay on any drive. You must, however, start AutoPlay manually when it is installed on a floppy disk. To do this, double-click the floppy disk icon, or right-click the floppy disk icon, and then click **AutoPlay**.
5. After you complete your tests of Autorun.inf, reset the value of NoDriveTypeAutoRun to 0000 95 00 00 00.

Important Because implementing AutoPlay on a floppy disk provides an easy way to spread computer viruses, it is appropriate to suspect that any publicly distributed floppy disk that contains Autorun.inf files is contaminated.

For more information about the NoDriveTypeAutoRun value, see Setting the NoDriveTypeAutoRun Value.

Setting the NoDriveTypeAutoRun Value

The NoDriveTypeAutoRun value in the registry is a 4-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 this byte 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. For information about how to change the registry values, see Using the Registry. 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 disk).
4	DRIVE_REMOTE	Network drive.
5	DRIVE_CDROM	CD-ROM drive.
6	DRIVE_RAMDISK	RAM disk.
7 (high-order bit)		Reserved for future use.

Note For Windows NT, you must restart Windows NT Explorer before any changes take effect.

Suppressing AutoPlay

There are a variety of reasons why you might want to suppress AutoPlay. One might be that your application has a setup program that requires the user to insert a disc which contains an Autorun.inf file. In this case, you would not want the AutoPlay feature to begin running an application while your setup application is running.

Another reason to suppress AutoPlay might be that your user needs to do disk swapping during your game. You would probably not want the setup program to execute while your game is in progress.

You can manually prevent the Autorun.inf file on a compact disc from being parsed and carried out by holding down the SHIFT key when you insert the disc.

Users of Windows NT version 4.0 can suppress AutoPlay automatically using the code example shown below. This initialization code is based on the assumption that your setup application is in the foreground window.

```
uMessage - RegisterWindowMessage(TEXT("QueryCancelAutoPlay"));
```

Then, add the following code to your setup window procedure:

```
if(msg == uMessage)
{
    // return 1 to cancel AutoPlay
    // return 0 to allow AutoPlay
    return 1L;
}
```

Windows 95 users can suppress AutoPlay by setting the value of the NoDriveTypeAutoRun registry entry to 0xFF. The code sample below demonstrates how this can be done. Insert this code near the beginning of the **WinMain** function.

```
const unsigned long WINDOWS_DEFAULT_AUTOPLAY_VALUE=0x095;
const unsigned long AUTOPLAYOFF=0x0FF;
     unsigned long ulOldAutoRunValue = WINDOWS_DEFAULT_AUTOPLAY_VALUE;
     unsigned long ulDisableAutoRun = AUTOPLAYOFF;
     unsigned long ulDataSize = sizeof(unsigned long);
     HKEY hkey = NULL;
char *lpzRegistryString = "Software\\Microsoft\\Windows\\CurrentVersion\
\Policies\\Explorer"

if( RegOpenKeyEx(HKEY_CURRENT_USER,
    lpzRegistryString ,
    0, KEY_ALL_ACCESS, &hkey ) == ERROR_SUCCESS )
{
    if( RegQueryValueEx(hkey,"NoDriveTypeAutoRun", 0, NULL,
                        (unsigned char*)&ulOldAutoRunValue,
                        &ulDataSize ) == ERROR_SUCCESS )
    {
        RegSetValueEx(hkey, "NoDriveTypeAutoRun", 0, REG_BINARY,
                      (const unsigned char*)&ulDisableAutoRun, 4);
    }
    else ulOldAutoRunValue = WINDOWS_DEFAULT_AUTOPLAY_VALUE;

    RegFlushKey( hkey );
    RegCloseKey( hkey );
}
```

Just before your application terminates, it must reset the registry to the old value, as shown in the code example below:

```
// Restore original AutoPlay settings.
if (RegOpenKeyEx(HKEY_CURRENT_USER,
                 lpzRegistryString ,
                 0, KEY_ALL_ACCESS, &hkey ) == ERROR_SUCCESS )
{
    RegSetValueEx(hkey, "NoDriveTypeAutoRun", 0, REG_BINARY,
                  (const unsigned char*)&ulOldAutoRunValue,
                   4 );
    RegFlushKey( hkey );
    RegCloseKey( hkey );
}
```

AutoPlay for MS-DOS-Based Applications

You also can use AutoPlay to install, configure, and run MS-DOS-based applications in a Windows MS-DOS session. You can even configure each MS-DOS-based application with its own unique icon, Config.sys file, and Autoexec.bat file.

Windows creates the correct configuration files for the MS-DOS-based application. The startup application then starts the MS-DOS-based application in a window.

AutoPlay Reference

This section contains reference information for the AutoPlay feature.

Commands

This section contains information about the following AutoPlay commands:

- **defaulticon**
- **icon**
- **open**
- **shell**
- **shell\verb**

defaulticon

The **defaulticon** command specifies an absolute path on the compact disc to the file that contains the information for the icon. The icon represents the AutoPlay-enabled CD in the Windows user interface.

```
defaulticon=path\iconname.ico
```

Parameters

path\iconname.ico

Absolute path and file name of the file containing the icon. The icon can be in a .ico, .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.

Remarks

If both the **icon** and **defaulticon** commands are present in an Autorun.inf file, AutoPlay uses the icon specified in the **defaulticon** command.

See Also

icon

icon

The **icon** command specifies a file that contains an icon which represents the AutoPlay-enabled CD in the Windows 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.

```
icon=filename.ico
```

Parameters

filename.ico

Name of the file containing the icon information. You also can 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.

Remarks

The following example specifies the second icon in a file to represent a compact disc. The first icon's index is set to zero.

```
icon=filename.exe 1
```

See Also

defaulticon

open

The **open** command specifies the path and file name of the application that AutoPlay runs when you insert the compact disc in a CD-ROM drive.

```
open=dir\filename.exe
```

Parameters

dir\filename.exe

Path and file name of any executable file to run when the compact disc is inserted. If no path is specified, Windows looks for the file in the root directory on the compact disc. Specify a relative path to locate the file in a subdirectory.

Remarks

Use the **open** command to open a startup application that provides instant feedback to the user. For more information about startup applications, see Opening a Startup Application.

AutoPlay can pass command-line parameters to the application when it runs. Specify command line parameters after the filename.

shell

The **shell** command changes the default entry of the shortcut menu to the specified custom command.

```
shell=verb
```

Parameters

verb

Abbreviated form of a custom command. The custom command must be defined in the Autorun.inf file.

Remarks

If the user right clicks the icon which represents your AutoPlay-enabled CD, a shortcut menu will be displayed. AutoPlay is the default menu item defined for any AutoPlay-enabled CD. The shell directive changes the default command to the specified command verb.

The command verb is executed when the user selects it from the shortcut menu. It is also executed when the user double-clicks the icon representing your CD.

See Also

shell\verb

shell\verb

The **shell\verb** command 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.

```
shell\verb\command=filename.exe
shell\verb=Menu Item Name
```

Parameters

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* on the shortcut menu unless *Menu Item Name* is omitted from the Autorun.inf file.

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 also can set a shortcut key for the menu item by preceding one of the letters in the item with an ampersand (&).

Remarks

When a user right-clicks an icon in the Windows user interface, a shortcut menu for that icon appears. If an Autorun.inf file is present on a CD and the user right-clicks the icon for the CD, Windows automatically adds AutoPlay to the shortcut menu for the disk's icon. It also sets AutoPlay as the default behavior. Double-clicking the icon starts whatever is specified in the **open** command.

To add the command **ReadMe** to the shortcut menu for your product and to make the letter "M" the shortcut key, include the following in the Autorun.inf file:

```
shell\readit\command=notepad abc\readme.txt
shell\readit=Read &Me
```

See Also

shell, **open**

Glossary

This chapter provides a glossary of technical terms used in the DirectX Programmer's Reference documentation. Entries are grouped by letter in alphabetical order.

A

alpha channel
The opacity of an image defined by an alpha value per pixel interleaved with the color components (for example, ARGB), an alpha value per pixel stored in a separate alpha surface, or a constant alpha value for the entire surface.

alpha color component
The portion of a 32-bit color that determines its opacity. In this case, the alpha value per pixel is interleaved with the color components (for example, ARGB). Less commonly, this term can also refer to an image with an alpha value per pixel stored in a separate alpha surface.

alpha constant
A level of opacity (alpha value) applied to an entire surface.

alpha edge blend
A particular use of alpha blending (and alpha channel information) to reduce aliasing by blending edges based on pixel coverage information.

ambient
A light source that illuminates everything in a scene, regardless of the orientation, position, and surface characteristics of the objects in the scene. Because this 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.

anisotropic filtering
A mipmap filtering mode that compensates for anisotropy, which is the distortion visible in the texels of a 3D object whose surface is oriented at an angle with respect to the plane of the screen. The anisotropy is measured as the elongation (length divided by width) of a screen pixel that is inverse-mapped into texture space.

array object
A group of objects organized into an array. 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.

attach
To connect multiple DirectDrawSurface objects into complex structures, like those needed to support 3-D page flipping with z-buffers. Attachment is not bidirectional, and a surface cannot be attached to itself. Emulated surfaces (in system memory) cannot be attached to nonemulated surfaces. Unless one surface is a texture map, the two attached surfaces must be the same size.

attached
Physically connected to the system. A device may be installed but not currently attached.

attack
The period at the beginning of a force feedback effect when the magnitude is reaching its basic or sustain level.

audio stream
Sound data, mixed or unmixed. DirectSound mixes audio streams from each secondary sound buffer that is playing and writes the result to the primary sound buffer, which supplies the sound hardware with audio data.

B

back buffer
A nonvisible surface that bitmaps and other images can be drawn to while the primary surface displays the currently visible image.

back clipping plane
The far boundary of a viewing frustum beyond which objects are not rendered. *See also* front clipping plane.

blend factor
The description of how each color component is blended in texture blending.

blend mode
The algorithm used to determine how a texture is blended with the colors of the surface to which the texture is applied.

blit
A bit block transfer.

bounds checking
The process of checking that an on-screen image is displayed within the bounds of the screen.

C

camera
A Direct3DRMFrame object used by the viewport to define 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.

clip list
A series of rectangles that describes the visible areas of the surface. The clip list cannot be set if a window handle is already associated with the DirectDrawClipper object.

clipper
A DirectDrawClipper object.

codec
Represents the phrase "compressor and decompressor."

collision detection
The process of determining if any pixels for two images share the same location on-screen. (See also, hit detection).

color key
A value indicating the color to be used for transparent or translucent effects. When using a hardware blitter, for example, all of the pixels of a rectangular area will be blitted except for the value that was set as the color key, thereby creating nonrectangular sprites on a surface.

color space
Any of several different methods of encoding and visualizing color. The two most common types of color space are RGB and YUV.

color-space conversion
A technique for converting a color in one color space to another color space. Typically, this conversion is from YUV colors from a video source to RGB for display.

color table
An array of *n* color values (normally RGB triples).

complex surface
The collective term for several DirectDrawSurface objects, all of which are attached to a root surface. The complex-surface structure can be destroyed only by destroying the root.

current play position
The location in a DirectSound buffer where a sound is being played.

current write position
The location in a DirectSound buffer where it is safe to change data in that buffer.

D

dark light
A light source that removes illumination from a scene, created by specifying negative values for the colors of the light.

dead zone
An area within the range of an axis where the axis is considered to be at the center.

decal
A texture that is rendered directly, as a visual. Decals are rendered into a viewport-aligned rectangle.

destination color key
The color that, in the case of blitting, will be replaced or, in the case of overlays, be covered up on the destination surface.

directional
A light source that 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. Directional light has orientation but no position, and it is commonly used to simulate distant light sources, such as the sun.

E

emissive property
The material property that determines whether a material emits light. The emissive property of a material is one of two properties that determines how the material reflects light. *See also* specular property.

emissive setting
See emissive property.

enable frame
The frame to which a light applies.

envelope
A set of values that defines the shape of a force feedback effect by modifying the magnitude at the beginning (attack) and end (fade).

even field
The second field comprising a video frame in an odd/even field set. Also known as field 2.

execute buffer
A fully self-contained, independent packet of information that describes a 3-D scene. An execute buffer contains 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.

F

face
A single polygon in a mesh.

fade
The period between the central, or sustain, portion of a force feedback effect and its end.

field
Data for one-half of a single video frame within a video stream. Each field contains data for every other scan line. Fields are sent and received in odd field, even field order.

field polarity
Quality determining whether a field is an even field or an odd field. Odd fields are defined when the trailing edge of VREF does not occur during a scan line. Even fields are defined as when the trailing edge of VREF occurs during a scan line.

flip
The process of swapping the addresses associated with the back and front buffers. This effectively swaps the image in the back buffer to the front buffer, thereby displaying the image.

flipping chain
A series of surfaces, attached to each other, that can be flipped. *See also* flip.

flipping surface
Any piece of memory that can be flipped. *See also* flip.

frame
An invisible box that provides a frame of reference for objects in a scene. Objects can be placed in a scene by specifying their spatial relationship to a relevant reference frame. Visual objects take their positions and orientations from frames. Also, a single image from a movie or animation.

front buffer
The first buffer in a flipping chain. In many cases, this will be the visible primary surface. In other cases, such as a flipping chain of textures, the front buffer is the surface the 3-D engine will get the texture from, but it is not the primary surface and is not displayed. In the case of flipping overlay surfaces, the front buffer is displayed, but is only a surface overlaid on the primary surface. *See also* primary surface.

front clipping plane
The near boundary of a viewing frustum. Any object closer to the camera than the front clipping plane is not rendered. The height of the front clipping plane defines the field of view. *See also* back clipping plane.

G

genlocking
The process of synchronizing one video signal with another. Because they are synchronized, the genlocked signal can be mixed with the original signal, allowing dissolves, wipes, and other transition effects.

group
A number of players arranged together as a set in a DirectPlay session.

H

HAL
The hardware-abstraction layer. Consists of hardware and device driver mechanisms that insulate applications from device-specific implementation details. If a capability requested by an application is not implemented by the current hardware, the capability will be emulated by the software.

hardware blitter
A hardware component, built into the display adapter, that performs efficient blit operations.

HEL
The hardware-emulation layer. Provides software-based emulation of features that are not present in hardware.

hit detection
See collision detection.

host
In DirectPlay, a virtual player whose ID is DPID_SYSMSG. System messages and messages sent to all players in a session are managed by the host.

HREF
Acronym for horizontal refresh. In a video stream, the HREF is active to signal that the display is to begin a new scan line. See also, VREF.

I

index palette
A DirectDrawPalette object whose entries are indices into another palette object.

L

latency
A delay in response; for example, the interval between the time that a sound buffer plays and the time that the speakers actually reproduce the sound. .

lobby client
Lobby management routines associated with the user's computer, including launching applications, updating the user interface, and communicating with the lobby server.

lobby server
Lobby management routines associated with a remote server. The lobby server coordinates all the information about the users connected to a specific application.

M

material
A property that determines 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.

mesh
A set of faces, each of which is described by a simple polygon.

mipmap
A sequence of textures, each of which is a progressively lower-resolution, prefiltered representation of the same image. A higher-resolution image is used when a visible object is close to the viewer; as the object moves farther away (and gets smaller), lower-resolution images are used.

mixing
In DirectSound, the process of combining sound buffers that are playing and writing 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.

Mode X
A hybrid display mode derived from the standard VGA Mode 13. This mode allows the use of up to 256KB of display memory (rather than 64KB allowed by Mode 13) by using the VGA display adapter's EGA multiple video plane system.

model coordinates
Coordinates that are relative to a child frame. *See also* world coordinates.

N

normal vector
An imaginary ray extending perpendicularly from a surface that defines the face's orientation.

O

odd field
The first field comprising a video frame in an odd/even field set.

off-screen surface
A conceptually rectangular area in memory that is generally used to store bitmaps that will be blitted to a back buffer before being displayed. Commonly used to store sprites.

opcode
Operation code that defines how the vertices in an execute buffer should be interpreted or how the state of the system should be changed.

overdraw
The average number of times to which a screen pixel is written.

overlay surface
A conceptually rectangular area in memory whose stored image information will cover the image information of the primary surface to which it is applied. Overlays are assumed to be on top of all other screen components.

overlay z-order
Determines the order in which overlays clip each other, enabling a hardware sprite system to be implemented under DirectDraw.

P

page flipping
See flip.

palette
The set of colors used by an object or application. In DirectX, a DirectDrawPalette object.

palette index
An integer index into the palette table array that is used to select a particular color.

pan value
The relative volume, measured in hundredths of decibels, between the left and right audio channels.

parallel point
A light source that illuminates objects with parallel light, but the orientation of the light is taken from the position of the parallel point light source. 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.

penumbra
The outer, dimly lit section of a spotlight's cone of light. The penumbra surrounds the umbra and merges with the surrounding deep shadow. *See also* umbra, *and* spotlight.

perspective correction
The technique of applying a texture map to a polygon that is angled away from the camera, interpolating so that the texture is stretched onto the polygon appropriately for the apparent depth of the polygon. Direct3D supplies perspective correction automatically.

pick
To search for visuals in a scene given a 2-D coordinate in a viewport.

pitch
The distance, in bytes, between an address that represents the beginning of a bitmap line and the beginning of the next line. Do not confuse memory pitch and memory width, since not all display memory is laid out as one linear block. For example, with rectangular memory, the pitch of the display memory could include the width of the bitmap plus part of a cache. *See also* width, stride.

player
A single participant in a DirectPlay session. Each player is associated with a player ID that enables messages to be exchanged among players.

player ID
A unique number that is assigned to each participant in a DirectPlay session when the participant is created. The application can exchange messages among players by using player IDs. The host is always assigned the DPID_SYSMSG player ID.

point
A light source that radiates equally in all directions from its origin.

power
In the specular property of a material, the value that determines the sharpness of specular highlights. A value of 5 gives a metallic appearance, and higher values give a more plastic appearance.

primary sound buffer
The buffer the user hears when a game is playing. The primary buffer is generally used to mix sound from secondary buffers, but it can be accessed directly for custom mixing or other specialized activities.

primary surface
The area in memory containing the image being displayed on the monitor. In DirectX, the primary surface is represented by the primary DirectDrawSurface object.

Q

quaternion
A fourth element added to the [*x*, *y*, *z*] values that define a vector. Quaternions define a 3-D axis and a rotation around that axis.

R

reference count

A control for a component object model (COM) object. When an object is created, its reference count is set to 1. Every time an interface is bound to the object, its reference count is incremented; when the interface connection is destroyed, the reference count is decremented. The object is destroyed when the reference count reaches 0. All interfaces to that object are then invalid.

root frame

A frame in Direct3D that has no parent frame; a frame at the top of a hierarchy of frames. The root frame contains the entire set of objects that make up a scene. *See also* scene.

S

saturation
Adjustment to the extreme value of a range when the actual value approaches the extreme. If the maximum value in a range is 1,000 and the maximum saturation point is set to 900, then any value greater than 899 is adjusted to 1,000.

scene
The entire set of objects that make up a virtual environment, including visible objects, sounds, lights, and frames. In Direct3D, the entire set of objects is contained by a root frame.

secondary sound buffer
A section of audio memory that stores individual sounds that are played throughout an application. The sound can be played as a single event or as a looping sound that plays repeatedly. Secondary buffers can also play sounds that are larger than available sound-buffer memory; the buffer serves as a queue that stores the portions of the sound about to be played.

service provider
A dynamic-link library used by DirectPlay to communicate over a network. The service provider contains all the network-specific code required to send and receive messages. Online services and network operators can supply service providers to use specialized hardware, protocols, communications media, and network resources.

session
In DirectPlay, an instance of several applications on remote machines communicating with each other.

sound buffer
A segment of memory that stores DirectSound audio data. Sound buffers can be primary or secondary, static or streaming.

source color key
A color that, in the case of blitting, will not be copied, or, in the case of overlays, not be visible on the destination.

specular property
The material property that determines how a point of light on a shiny object corresponds to the reflected light source. The specular property of a material is one of two properties that determines how a material reflects light. *See also* emissive property.

spotlight
A light source that 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.

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

sticky focus
In DirectSound, the capability to play sound buffers when the owning application does not have the input focus. For example, a DirectSound application could continue to play a sound buffer while the user was working in another application.

streaming sound buffer
A small sound buffer that can play lengthy sounds because the application dynamically loads audio data into the buffer as it plays. For example, an application could use a buffer that can hold 3 seconds of audio data to play a 2-minute sound. A streaming buffer requires much less memory than a static buffer.

stretching
Blitting an image into a destination with different dimensions. This operation is supported directly by

some hardware.

stride
Synonymous with pitch. *See also* width.

surface
Memory that represents visual images. This is often display memory, but it can be system memory. *See also* complex surface, off-screen surface, overlay surface, *and* primary surface.

sustain
The period when the basic magnitude of a force feedback effect is attained, after the attack and before the fade.

T

tearing
A visual artifact produced when the screen refresh rate is out of sync with an application's frame rate. The top portion of one frame is displayed at the same time as the bottom portion of another frame, with a discernible tear between the two partial images.

texel
A single element in a texture. When a texture has been applied to an object, the texels rarely correspond to pixels on the screen. Applications can use texture filtering to control how texels are interpolated to pixels.

texture
A rectangular array of pixels that is applied to a visual object in Direct3D.

texture blending
The technique of combining the colors of a texture with the colors of the surface to which the texture is applied.

texture coordinates
The coordinates that determine which texel in each texture is assigned to each vertex in an object.

texture mapping
The application of a texture to an object. Because a texture is a flat image and the object is often not, the texture must be mapped to the surface of the object, using texture coordinates and wrapping flags. *See also* texture coordinates *and* wrap.

U

umbra

The central brightly lit section of a spotlight's cone of light. The umbra can act as a point source. *See also* penumbra, *and* spotlight.

V

VBI
Abbreviation for Vertical Blanking Interval.

vertex
A point in 3-D space.

video frame
A single image in a video stream, comprised of one odd field and one even field. *See also,* field.

viewing frustum
A 3-D volume in a scene positioned relative to the viewport's camera. Objects within the 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. For orthographic viewing, the viewing frustum is cuboid.

viewport
A rectangle that defines how a 3-D scene is rendered into a 2-D window. A viewport also defines an area on a device into which objects will be rendered.

VREF
Acronym for vertical refresh. In a video stream, the VREF is active to signal that the display is to begin a new screen. *See also,* HREF.

W

width
The distance between two addresses in memory that represent the beginning of a line and the end of the line of a stored bitmap. This distance represents only the width of the bitmap in memory; it does not include any extra memory required to reach the beginning of the next line of the bitmap, such as a cache in rectangular memory.

world coordinates
Coordinates that are relative to the root frame. *See also* model coordinates.

wrap
The procedure used to calculate texture coordinates for a face or mesh. The basic wrapping types are flat, cylindrical, spherical, and chrome.

Z

z-buffer
A buffer that stores a depth value for each pixel in a scene. Pixels with a small z-value overwrite pixels with a large z-value.