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 DirectX Programmer's Reference
- DirectX and the Component Object Model
- What's New in the DirectX 5 Programmer's Reference?
- Conventions

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
- Detecting DirectX Versions
- Using Macro Definitions

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
- IUnknown Interface
- DirectX COM Interfaces
- C++ and the COM Interface
- Retrieving Newer Interfaces
- Accessing COM Objects by Using C
- Interface Method Names and Syntax

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.