Chapter 1

Microsoft® DirectX™ 3 Software Development Kit

Introducing DirectX 3

Introducing DirectX 3

CHAPTER 1

DirectX Goals

The Microsoft® DirectX™ Software Development Kit (SDK) 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 because it wanted the performance of applications running in the Microsoft Windows® operating system to rival or exceed the performance of applications running in the MS-DOS® operating system or on game consoles. This SDK 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.

This section discusses two important benefits of using DirectX: providing hardware independence for software developers and setting guidelines for hardware developers.

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

Benefits of Developing DirectX Windows Applications

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.

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.

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

When Microsoft created DirectX, one of its primary goals was to promote games development for the Windows operating 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.

Another important goal was to provide guidelines for hardware companies based on feedback from developers of high-performance applications and independent hardware vendors (IHVs). As a result, the DirectX SDK 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.

Display-hardware features that will be available soon include:

- Overlays, which will be supported so *page flipping* will be enabled within a window in a graphic device interface (GDI). Page flipping is the double-buffer scheme used to display frames on the entire screen.
- Sprite engines, which make overlaying sprites easier.
- Stretching with interpolation, which efficiently conserve display memory because it stretches a smaller frame to fit the entire screen.
- Alpha blending, which mixes colors at the hardware-pixel level.
- Three-dimensional (3D) accelerators with perspective-correct textures, which allow you to display textures on a 3D surface. For example, you can texture hallways in a castle generated by 3D software with a brick-wall bitmap that maintains the correct perspective.
- Blits for 3D graphics that take z-buffers into account.
- Standard 2 megabytes (MB) of display memory, which is typically the minimum required by 3D games.
- Compression standard, which allows you to store more data in display memory. This standard will be very fast when implemented in either software or hardware. It will be used for textures and will include transparency compression.

Audio-hardware features that will be available soon include:

- Hardware and enhancers that provide a 3D spatial placement for different sounds.
- On-board memory for audio boards.
- Audio-video combination boards that share on-board memory.

In addition, video playback will benefit from future DirectX-compatible hardware accelerators. One of the features that future releases of DirectX will support is hardware-accelerated decompression of YUV video.

The DirectX SDK

This section describes the DirectX SDK and some DirectX implementation details. The following topics are discussed:

- *DirectX SDK Components*
- *Using Macro Definitions*

DirectX SDK Components

The DirectX SDK 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 to the chapter for 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 about this component, see *About DirectDraw* in the DirectDraw documentation.
- DirectSound® enables hardware and software sound mixing and playback. For more information about this component, see *About DirectSound* in the DirectSound documentation.
- DirectPlay® makes connecting games over a modem link or network easy. For more information about this component, see *About DirectPlay* in the DirectPlay documentation.
- Direct3D™ provides a high-level Retained-Mode interface that allows applications to easily implement a complete 3D graphical system, and a low-level Immediate-Mode interface that lets applications take complete control over the rendering pipeline. For more information about this component, see *About Direct3D* in the Direct3D documentation.
- DirectInput™ provides input capabilities to your game that are scalable to future Windows-based hardware-input APIs and drivers. Currently the joystick, mouse, and keyboard are supported. For more information about this component, see *Introduction to Joysticks* in the DirectInput documentation.
- DirectSetup provides a one-call installation procedure for DirectX. For more information about this component, 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 about this component, see *About AutoPlay* in the AutoPlay documentation.

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

Among the most important parts of the documentation for the DirectX SDK 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 SDK.

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*
- *Accessing COM Objects by Using C*
- *Interface Method Names and Syntax*

The Component Object Model

Most APIs in the DirectX SDK 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 OLE programming. It is also an interface specification from which any number of interfaces can be built. It is an object model at the operating-system level.

Many DirectX APIs are instantiated as a set of OLE *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 OLE 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 OLE 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 *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.

AddRef and **Release** 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.

QueryInterface 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

```
ULONG AddRef();
```

Increases the object's reference count by 1.

- Returns the new reference count.

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

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

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.

- Returns S_OK if the call succeeds. If the call fails, the method returns 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** only)

  DirectSound

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

  DSERR_INVALIDPARAM

  DirectPlay

  DPERR_INVALIDOBJECT

  DPERR_INVALIDPARAMS

For Direct3D's 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*.

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.

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

```
ULONG Release();
```

Decreases the object's reference count by 1.

- Returns the new reference count.

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 SDK 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* OLE interface. Creation of these basic objects is handled by specialized functions in the dynamic link library (DLL) for each object, rather than by the Win32 **CoCreateInstance** function typically used to create COM objects.

Typically, the DirectX SDK 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++.

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 3 SDK?

The DirectX 3 SDK provides more services—and more avenues for innovation—than did the DirectX 2 SDK. Although this SDK contains additional functions and services, all the applications you wrote with the DirectX 2 SDK, or the original DirectX 1 SDK, will compile and run successfully without changes.

The purpose of this section is to help those of you who are familiar with the DirectX 2 SDK quickly identify several important areas of this SDK that are significantly different. These differences are listed by component.

DirectDraw

No changes to the API. The documentation has been updated to include a series of tutorials that provide step-by-step instructions for implementing a simple DirectDraw application. To read these tutorials, see *DirectDraw Tutorials* in the DirectDraw documentation.

DirectSound

DirectX 3 contains DirectSound3D functionality, which enables an application to change the apparent position of a sound source. Applications can specify sound cones for directional sound sources, Doppler-shift effects for moving sounds, and distances at which different effects occur. For more information about this new feature of DirectSound, see *Three-Dimensional Sound* in the DirectSound section of the documentation.

DirectPlay

DirectPlay has become a technology family that provides not only a way for applications to communicate with each other that is independent of the underlying transport, protocol, or online service, but also the independence for matchmaking servers. The *IDirectPlay2*, *IDirectPlay2A*, and *IDirectPlayLobby* interfaces were added to implement this new technology. For more information about what's new in DirectPlay, see *What's New in DirectPlay Version 3?*.

Direct3D

No changes to the API. The Retained-Mode tutorial has been updated and simplified. To read this tutorial, see *Direct3D Retained-Mode Tutorial* in the Direct3D documentation.

DirectInput

DirectInput now includes support for mouse and keyboard input devices, as well as joysticks.

DirectSetup

DirectSetup has a new function that helps applications make the proper entries in the registry during installation.

AutoPlay

The AutoPlay documentation now includes Windows NT® information.

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.
[]	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.