Microsoft® DirectX™ 2 Software Development Kit

C H A P T E R 1

Introducing the DirectX 2 Software Development Kit

Using DirectX 2 in Windows

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

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

Reasons for Developing DirectX Windows Applications

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

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

A high-performance Windows-based game will:

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

Providing Standards for Hardware Accelerators

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

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

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

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

Audio hardware features that will be released include:

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

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

DirectX 2 Components

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

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

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

DirectDraw

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

A DirectDraw object, created using the **DirectDrawCreate** function, represents the display adapter card. One of the object's methods, **IDirectDraw::CreateSurface**, creates the primary DirectDrawSurface object,

which represents the display memory being viewed on the monitor. From the primary surface, off-screen surfaces can be created in a linked-list fashion.

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

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

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

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

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

Finally, DirectDraw supports overlays in hardware and by software emulation. Overlays present an easier means of implementing sprites and managing multiple layers of animation. Any DirectDrawSurface object can be created as an overlay with all of the capabilities of any other surface, plus extra capabilities associated only with overlays. These capabilities require extra display memory, and if there

are no overlays in display memory, the overlay surfaces can exist in system memory.

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

DirectSound

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

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

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

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

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

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

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

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

DirectPlay

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

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

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

Direct3D

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

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

mapping, hardware 3D rendering in 2D overlay planes—and even sprites—providing seamless use of 2D and 3D graphics in interactive media titles.

DirectInput

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

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

AutoPlay

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

Sample Applications

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

DirectX and the Component Object Model

The Component Object Model

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

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

interface to get the current display mode of the display adapter from the DirectDraw object.

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

IUnknown

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

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

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

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

DirectX 2 SDK COM Interfaces

The interfaces in the DirectX 2 SDK have been created at a very basic level of the COM programming hierarchy. Each main device object interface, such as IDirectDraw, IDirectSound, or IDirectPlay, derives directly from the IUnknown interface in OLE. Creation of these basic objects is handled by specialized

functions in the dynamic link library (DLL) for each object rather than by the Win32 **CoCreateInstance** function typically used to create COM objects.

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

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

C++ and the COM Interface

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

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

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

Accessing COM Objects Using C

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

- The first parameter of the method is always a reference to the object that has been created and is invoking the method (the *this* argument).

- Each method in the interface is referenced through a pointer to the object's vtable.

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

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

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

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

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

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

Interface Method Names and Syntax

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

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

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

The same example using C syntax looks like this:

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

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

Using Macro Definitions

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

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

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

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

Floating-point Precision

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

Differences Between the Game SDK and the DirectX 2 SDK

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

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

DirectDraw

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

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

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

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

  DDERR_CANTLOCKSURFACE **DDERR_CANTPAGELOCK**
  DDERR_CANTPAGEUNLOCK **DDERR_DCALREADYCREATED**
  DDERR_INVALIDSURFACETYPE **DDERR_NOMIPMAPHW**
  DDERR_NOTPAGELOCKED **DDERR_NOTINITIALIZED**

- The **IDirectDraw2::SetDisplayMode** method contains two new parameters, *dwRefreshRate* and *dwFlags*. If neither of these parameters are needed, you can still use **IDirectDraw::SetDisplayMode**.
- The **IDirectDraw2::EnumDisplayModes** method was added to enumerate the refresh rate of the monitor and store it in the **dwRefreshRate** member of the **DDSURFACEDESC** structure.
- A new method has been added to the IDirectDraw2 interface: **IDirectDraw2::GetAvailableVidMem**.
- Three new methods have been added to the IDirectDrawSurface2 interface: **IDirectDrawSurface2::GetDDInterface**, **IDirectDrawSurface2::PageLock**, and **IDirectDrawSurface2::PageUnlock**.
- DirectDraw in the Game SDK limited the available display modes to 640 by 480 with pixel depths of 8 bpp and 16 bpp. DirectDraw now allows an application to change the mode to allow any supported by the display driver.
- DirectDraw now checks the display modes it is capable of using against the display restrictions of the installed monitor. If the requested mode is not

compatible with the monitor, then the **IDirectDraw2::SetDisplayMode** method will fail. Only modes supported on the installed monitor will be enumerated in the **IDirectDraw2::EnumDisplayModes** method.

- In the Game SDK, DirectDraw only allowed the creation of one DirectDraw object per process. If your process happened to use another system component, such as DirectPlay, that created a DirectDraw object, the process would be unable to create another DirectDraw object for its own use. It is now possible for a process to call the **DirectDrawCreate** function as many times as necessary. A unique and independent interface will be returned from each call. For more information, see **Multiple DirectDraw Objects per Process**.
- DirectDraw on the Game SDK required an HWND be specified in the **IDirectDraw::SetCooperativeLevel** method call regardless of whether or not a full-screen, exclusive mode was being requested. This method no longer requires an HWND be specified if the application is requesting DDSCL_NORMAL mode. It is now possible for an application to use DirectDraw with multiple windows. All of these windows can be used simultaneously in normal mode.
- In DirectDraw on the Game SDK, if a surface was in system memory, the hardware emulation layer (HEL) automatically performed the blit. However, some display cards have DMA hardware that allows them to efficiently blit to and from system memory surfaces. In the DirectX 2 version of DirectDraw, the **DDCAPS** structure has been expanded to allow drivers to report this capability. The following members have been added:

```
DWORD       dwSVBCaps
DWORD       dwSVBCKeyCaps
DWORD       dwSVBFXCaps
DWORD       dwSVBRops[DD_ROP_SPACE]

DWORD       dwVSBCaps
DWORD       dwVSBCKeyCaps
DWORD       dwVSBFXCaps
DWORD       dwVSBRops[DD_ROP_SPACE]

DWORD       dwSSBCaps
DWORD       dwSSBCKeyCaps
DWORD       dwSSBFXCaps
DWORD       dwSSBRops[DD_ROP_SPACE]
```

- In DirectDraw on the Game SDK, palettes could only be attached to the primary surface. Palettes can now be attached to any paletized surface (primary, back buffer, off-screen plain, or texture map). For more information, see **Setting Palettes on Non-Primary Surfaces**.
- Palettes can now be shared between multiple surfaces. For more information, see **Sharing Palettes**.

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

DirectSound

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

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

DirectPlay

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

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

Direct3D

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

DirectInput

No changes were made to this version of DirectInput.

AutoPlay

No changes were made to this version of AutoPlay.

DirectSetup

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

Conventions

The following conventions are used to define syntax.

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

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

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