Introducing the Game Software Development Kit

Introducing the Game Software Development Kit

Introducing the Game Software Development Kit

Introducing the Game Software Development Kit

Programming Games in Windows

This topic provides an introduction to the components of the Microsoft Game Software Development Kit (SDK). It includes summary information about the features and background information on topics such as the component object model (COM) programming and the AutoPlay feature of the Microsoft Windows 95 operating system.

Microsoft developed the Game SDK for several reasons. The primary reason is to make performance on the Windows-based platform rival or exceed performance on MS-DOS-based platforms and console-system platforms.

Another reason to promote game development in Windows is to assist game developers by providing a robust, standardized, and well-documented platform for which to write games.

Reasons for Developing Windows-Based Games

Why should there be interest in games for the personal computer and Windows 95? 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. With video performance as strong as the best console or MS-DOS-based game, the Windows-based game becomes a true contender for market share.

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.

Microsoft provides a consistent interface between hardware manufacturers and game developers to reduce the complexity of installation and configuration, and to utilize the hardware to its best advantage. MS-DOS-based games can take advantage of the same hardware cards available to the Game SDK developer. However, when developing MS-DOS-based games, a developer must conform to varying implementations of cards which complicates installation.

Providing Standards for Hardware Accelerators

The primary goals of the Game 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 that are based on feedback from game developers and independent hardware vendors (IHVs). Therefore, the Game SDK components often provide specifications for hardware accelerator features that do not yet exist. In many cases, these specifications are emulated in 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 video hardware features coming out in the near future include:

- Overlays. Overlays will be supported so that page flipping will also be enabled within a window in 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 video RAM.
- Alpha blending, used to mix colors at the hardware pixel level.
- Three-dimensional (3-D) accelerators with perspective-correct textures. This feature will allow textures to be displayed on a 3-D surface so; for example, hallways in a castle generated by 3-D software can be textured with a brick wall bitmap that maintains the correct perspective.
- Z buffer-aware bit-block transfers for 3-D graphics.
- 2 megabytes (MB) of video memory is standard. 3-D games generally need at least this much video RAM.
- A compression standard so you can put more data into display memory. This standard will include transparency compression, be usable for textures, and be very fast when implemented in software as well as hardware.

Add audio hardware features such as:

- 3-D audio enhancers that provide a spatial placement for different sounds (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 Game SDK. One of the features that will be supported by future releases of the Game SDK is hardware-accelerated decompression of YUV video.

Windows 95 Game Features

The Game SDK is composed of several interfaces that address and answer the performance issues of programming games 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 offscreen video memory as well as extremely fast access to the bit-block transferring 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 DirectInput application programming interface. This provides joystick input capabilities to your game that are scalable to future Windows hardware input API and drivers.
- AutoPlay, which 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 Game SDK.

Each feature of the Game SDK is described in more detail in its corresponding section.

DirectDraw

The biggest gain in performance in the Game SDK comes from the DirectDraw services, which are a combination of four COM interfaces: **IDirectDraw**, **IDirectDrawSurface**, **IDirectDrawPalette**, and **IDirectDrawClipper**. For more information about the COM concepts required for programming games using the Game SDK, see the section Games and the Component Object Model.

A DirectDraw object, created using the function **DirectDrawCreate**, represents the video display card. One of the object's member functions, **IDirectDraw::CreateSurface** creates the primary DirectDrawSurface object, which represents the video display memory being viewed on the monitor. From the primary surface, offscreen 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 is used to exchange 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 composed by transferring a series of offscreen bitmaps stored on other DirectDrawSurface objects in video RAM. Call the **DirectDrawSurface::Flip** member function to display the recently composed frame, which sets a register so that the exchange occurs when the screen performs a vertical retrace. This is asynchronous, so the game can continue processing after calling **DirectDrawSurface::Flip.** (The back buffer is automatically write-blocked after calling **DirectDrawSurface::Flip** until the exchange occurs.) After the exchange occurs, the game continues to compose the next frame in the back buffer, call **DirectDrawSurface::Flip**, and so on.

DirectDraw improves performance over the Windows 3.1 GDI model, because in the latter, there is no direct access to bitmaps in video memory. Therefore, bit-block transfers always occur in host RAM and are then transferred to video display memory. Using DirectDraw, all processing is done on the card whenever possible. DirectDraw improves performance over the Windows 95 and Microsoft Windows NT GDI model that 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 video hardware, there is an object that 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 video hardware. DirectDrawPalette objects are also created from DirectDraw objects.

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

Transparent bit-block transfers 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 bit-block transfers 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 and any overlay has 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 host memory.

Color keying works the same for overlays as for transparent bit-block transfers. The Z order of the overlay automatically handles the occlusion and transparency manipulations between overlays.

DirectSound

Programming for 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 Game SDK, use the hardware to its greatest advantage whenever possible and emulate 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 games using the Game SDK, see the section Games and the Component Object Model.

The DirectSound object represents the sound card and its various attributes. The DirectSoundBuffer object is created using a DirectSound object and represents a buffer containing sound data. Several DirectSoundBuffer objects can exist and can 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 host 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 that are played throughout a game. A sound stored in a secondary buffer can be played as a single event or as a looping sound that plays repeatedly.

You can use secondary buffers to play sounds that are larger than available sound buffer memory. When used to play a sound that is larger than the buffer, the secondary buffer is a queue that stores the portions of the sound that are about to be played.

DirectPlay

One of the most compelling features of the personal computer as a game platform is the easy access to communication services provided by desktop computers. 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.

DirectInput

The joystick represents a class of devices that report tactile movements and actions that 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 track balls and flight harnesses.

DirectInput is currently 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 the feature of Windows 95 that automatically plays a CD or audio CD when inserted into a CD-ROM drive. While this feature is not specific to the Windows 95 Games 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 Game SDK are located in the Sdk\Samples directory.

Games and the Component Object Model

The Game SDK is composed of several interfaces which are based on the component object model (COM). The interface member functions can be treated exactly like C functions.

This section provides enough information to understand COM in the perspective of the interfaces used in the Game SDK. It also demonstrates how to call member functions defined by an interface from C, and what is going on when this function occurs.

The Component Object Model

COM is a foundation for an object-based system that focuses on the reuse of interfaces. It is the model at the heart of OLE programming, not to be confused with OLE 2.0. COM is an interface specification from which any number of interfaces can be built. It is an object model at the operating system level.

COM interfaces are sometimes called *classes*. Each interface supports a set of functions that can perform services for the object. Other interfaces may exist that inherit the definitions of the methods from any particular interface. Unlike C++ or other object-oriented languages, only the *definition* for methods is inherited, not the code. It is up to the object that implements a COM interface to support all methods advertised by that interface and any interfaces it derives from.

Objects can bind to other objects at run-time and use the implementation of interfaces provided by the other object. By knowing that an object is a COM object, and knowing what interfaces it supports, a program or another object can determine what services that object can be called upon to perform. Therefore, one of the member functions inherited by all COM objects, called **QueryInterface**, lets you determine what interfaces are supported by the object and create pointers to these interfaces.

All COM interfaces are derived from an interface called **IUnknown**. **IUnknown** has only three member functions:

- **QueryInterface**, which determines if an interface is present.
- **AddRef**, which adds a reference for each object that is created or newly associated with another object.
- **Release**, which decrements the reference count for an object that no longer exists or is no longer associated with another object.

Note that with objects, **AddRef** is automatically called by the function that creates the object. However, the application must call **Release** whenever that object is destroyed. Otherwise, memory leaks can occur.

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 there is no state data associated with the interface. In a C++ abstract base class, all member functions 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 of the functions that implement the given interface. A program or object that wants to use these functions can use the **QueryInterface** function to verify that the interface exists on an object, and to obtain a pointer to that interface. What the program or object actually receives from the object after sending **QueryInterface** is a pointer to the vtable, through which it can call the interface functions 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 function 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 member function can be called from a C program. There are two things you need to remember when calling an interface member function from C:

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

The following example creates a surface associated with a DirectDraw object using the **IDirectDraw::CreateSurface** member function. The DirectDraw object that is associated with the new surface is referenced by the *lpDD* parameter. Incidentally, this member function fills in a surface description structure (*&ddsd*) and returns a pointer the new surface (*&lpDDS*).

The example calls **IDirectDraw::CreateSurface** using the C programming language. Notice that to call **IDirectDraw::CreateSurface**, you dereference the DirectDraw object's vtable and then dereference the function from the vtable. The first function parameter is a reference to the DirectDraw object that has been created and is invoking the function:

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

DirectDraw includes macro definitions to simplify usage of the member functions. The following example uses the macro to call **IDirectDraw::CreateSurface**. The first function parameter is a reference to the DirectDraw object that has been created and is invoking the function:

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

To illustrate the difference between calling a COM object method in C and C++, the same function in C++. C++ implicitly dereferences the vtbl pointer and passes the *this* pointer:

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

Note

Macro definitions for the member functions of each **IDirectDraw**, **IDirectPlay**, and **IDirectSound** interfaces are included in the header files of the respective components.

Game SDK COM Interfaces

The interfaces in the Game SDK have been created at a very base level of the COM programming hierarchy. Each main device object interface, such as **IDirectDraw**, **IDirectSound**, or **IDirectDraw** derives directly from **IUnknown**. The creation of these base objects is handled by specialized functions in the library rather than by the Win32 **CoCreateInstance** function normally used to create COM objects.

The Game 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 video 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.

Conventions

The following conventions are used to define syntax.

Convention	Meaning
Bold text	Denotes a term or character to be typed literally, such as a predefined data type or function name (**HWND** or **DirectDrawCreate**), a command, or a command-line option (**/x**). You must type these terms exactly as shown.
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.