# Introducing DirectX 7.0

[This is preliminary documentation and subject to change.]

The Microsoft® DirectX® 7.0 application programming interface (API) 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 7.0 Programmer's Reference in the Platform Software Development Kit (SDK) documentation. Information is divided into the following sections:

- About DirectX Help
- DirectX Goals
- The DirectX Programmer's Reference
- DirectX and the Component Object Model
- What's New in DirectX?
- Conventions
- Further Reading
- Debugging DirectX Applications
- Compiling DirectX Samples and Other DirectX Applications
- About DirectX for Visual Basic

# About DirectX Help

[This is preliminary documentation and subject to change.]

This documentation includes information about developing Microsoft® DirectX® applications in C, C++, and Microsoft Visual Basic®.

DirectX fully supports both C and C++. Although this document most often uses the C++ syntax, some of the examples and tutorials (as well as some of the sample applications) are in C.

In addition, the following topics provide general information on using C with DirectX:

- Using Macro Definitions
- Accessing COM Objects by Using C

For the sake of simplicity, this documentation generally uses "C++" to mean "C or C++" when presenting information for those languages as opposed to Visual Basic. You should assume that discussions of C++ concepts are also valid for C, with the necessary changes in syntax.

In order to present you only with the information that is relevant to your programming environment, this document implements language filtering. On the title bar of many topics you will find a pop-up language menu. This menu gives you the choice of seeing documentation tailored either to C++ (and C) or to Visual Basic. If you choose Show All, you will see the language-specific information for both languages. Regardless of your language selection, you will always see information (such as general concepts) that is relevant to all languages.

# DirectX Goals

[This is preliminary documentation and subject to change.]

Microsoft® DirectX® 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

[This is preliminary documentation and subject to change.]

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

[This is preliminary documentation and subject to change.]

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 is preliminary documentation and subject to change.]

This section describes the DirectX Programmer's Reference, also referred to as the DirectX Software Development Kit (SDK), components and some DirectX implementation details. The following topics are discussed:

- DirectX Programmer's Reference Components
- Using Macro Definitions
- Using Callback Functions

## DirectX Programmer's Reference Components

[This is preliminary documentation and subject to change.]

The DirectX Programmer's Reference, also known as 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 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.
- DirectMusic® is the musical component of DirectX. Unlike DirectSound, which is for the capture and playback of digital sound samples, DirectMusic works with message-based musical data. For more information about this newest component of DirectX, see About DirectMusic.
- 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 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 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 \Mssdk\Samples\Multimedia folder of the DirectX SDK or in the DirectX code samples under the Platform SDK References section.

# Using Macro Definitions

[This is preliminary documentation and subject to change.]

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, and also have the advantage of expanding to appropriate calls in either C or C++ syntax, depending on whether or not __cplusplus is defined.

The following example uses the **IDirectDraw4_CreateSurface** macro to call the **IDirectDraw4::CreateSurface** method. The first parameter is a reference to the DirectDraw object that has been created and invokes the method:

```
ret = IDirectDraw4_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.

# Using Callback Functions

[This is preliminary documentation and subject to change.]

DirectX contains many enumeration methods that are used to iterate through hardware resources or other items available to the application. These methods all work in fundamentally the same way.

Prototype callback functions are documented for each DirectX component. The application declares its own functions with the same return values and parameters as these prototypes. Callback functions are declared as type **CALLBACK**, **WINAPI**, or **FAR PASCAL**, which are all equivalent.

When the application calls an enumeration method, it supplies a pointer to the appropriate callback function that it has implemented. The method calls the function once for each item that qualifies for enumeration, and passes in information about the item. Within the function the application can use this information to perform any sort of task, such as building a list or looking for particular device capabilities. The enumeration method returns when all items have been enumerated or when the callback function returns a value indicating that enumeration can stop (either FALSE or a particular **HRESULT**, depending on the return type of the callback).

For an example, see DirectInput Device Enumeration.

# DirectX and the Component Object Model

[This is preliminary documentation and subject to change.]

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

# The Component Object Model

[This is preliminary documentation and subject to change.]

Most of the DirectX application programming interface is composed of objects and interfaces based on COM. COM is a foundation for an object-based system that focuses on reuse of interfaces. It is also an interface specification from which any number of interfaces can be built.

A DirectX application is built from instances of COM *objects*. You can consider an object to be a black box that represents hardware or data which you can access through *interfaces*. Commands are sent to the object through *methods* of the COM interface. For example, the **IDirectDraw4::GetDisplayMode** method of the **IDirectDraw4** interface is called 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 a 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

[This is preliminary documentation and subject to change.]

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

[This is preliminary documentation and subject to change.]

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

```
ULONG AddRef();
```

There are no parameters.

## Return Values

Returns the new reference count. This value is meant to be used for diagnostic and testing purposes only.

## 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.

## QuickInfo

**Windows NT/2000:** Requires Windows NT 3.1 or later.
**Windows 95/98:** Requires Windows 95 or later.
**Windows CE:** Unsupported.
**Header:** Declared in unknwn.h.

# IUnknown::QueryInterface

[This is preliminary documentation and subject to change.]

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
);
```

*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 may be E_NOINTERFACE, E_POINTER, or one of the following interface-specific error values. Interface-specific error values are listed by component.

DirectDraw
DDERR_INVALIDOBJECT
DDERR_INVALIDPARAMS
DDERR_OUTOFMEMORY (DirectDrawSurface objects only)

DirectSound
DSERR_GENERIC (**IDirectSound** and **IDirectSoundBuffer** only)
DSERR_INVALIDPARAM
DSERR_NOINTERFACE

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.

### QuickInfo

**Windows NT/2000:** Requires Windows NT 3.1 or later.
**Windows 95/98:** Requires Windows 95 or later.
**Windows CE:** Unsupported.
**Header:** Declared in unknwn.h.

# IUnknown::Release

[This is preliminary documentation and subject to change.]

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

```
ULONG Release();
```

There are no parameters.

### Return Values

Returns the new reference count. This value is meant to be used for diagnostic and testing purposes only.

### 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.

Applications must only call this method to release interfaces it explicitly created in a previous call to **IUnknown::AddRef**, **IUnknown::QueryInterface**, or a creation function such as **DirectDrawCreate**.

### QuickInfo

**Windows NT/2000:** Requires Windows NT 3.1 or later.
**Windows 95/98:** Requires Windows 95 or later.
**Windows CE:** Unsupported.
**Header:** Declared in unknwn.h.

# C++ and the COM Interface

[This is preliminary documentation and subject to change.]

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

[This is preliminary documentation and subject to change.]

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 follow this philosophy. For example, the DirectDraw component supports three interfaces to access a DirectDrawSurface object: **IDirectDrawSurface, IDirectDrawSurface2**, **IDirectDrawSurface3** and **IDirectDrawSurface4**. 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

[This is preliminary documentation and subject to change.]

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 **IDirectDraw4::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 **IDirectDraw4::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

[This is preliminary documentation and subject to change.]

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 **IDirectDraw4::GetCaps** method:

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

The same example using C syntax looks like this:

```
HRESULT GetCaps(
  LPDIRECTDRAW4 lpDD,
  LPDDCAPS lpDDDriverCaps,
  LPDDCAPS lpDDHELCaps
);
```

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

# What's New in DirectX?

[This is preliminary documentation and subject to change.]

Information for this topic is unavailable for this preliminary release of DirectX, but will be completed for the final release.

# Conventions

[This is preliminary documentation and subject to change.]

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** |
|---|---|
| FULL CAPITALS | Indicates most type and structure names, which also are bold, and constants. |
| monospace | Sets off code examples and shows syntax spacing. |

# Further Reading

[This is preliminary documentation and subject to change.]

Further explanation of the graphics and multimedia concepts and terms discussed throughout DirectX, as well as information on Windows programming in general, can be found in the following sources:


- Bargen, Bradley and Peter Donnelly, *Inside DirectX*, Microsoft Press, 1998.
- Begault, Durand R., *3-D Sound for Virtual Reality and Multimedia*, Academic Press, 1994.
- Blinn, James, *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann, 1996.
- Foley, James D., *Computer Graphics: Principles and Practice*, Addison-Wesley, 1991 (2nd edition).
- Hearn, Donald and M. Pauline Baker, *Computer Graphics*, Prentice-Hall, 1986.
- Kientzle, Tim, *A Programmer's Guide to Sound*, Addison-Wesley Developers Press, 1998.
- Petzold, Charles, *Programming Windows 98*, Microsoft Press, 1998 (5th edition).
- Thompson, Nigel, *3D Graphics Programming for Windows 95*, Microsoft Press, 1996.

- Watt, Alan H., and Mark Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992.


Additional sources for the concepts and terms associated with COM can be found in the following sources:


- Brockschmidt, Kraig, *Inside OLE 2*, Microsoft Press, 1995 (2nd edition).
- Rogerson, Dale E., *Inside COM,* Microsoft Press, 1997.


# Debugging DirectX Applications

[This is preliminary documentation and subject to change.]

When using debug builds, in order to ensure that the debugger can find all the relevant symbolic information, the symbol files need to be located as follows:

| OS | Debugger | Location for .pdb file | Location for .db |
|---|---|---|---|
| Windows 95<br>Windows 98 | Visual C++® | same directory as binary | same directory as |
| Windows NT<br>Windows 2000 | Visual C++ | same directory as binary | %SystemRoot%\ |
| Windows NT<br>Windows 2000 | NTSD/KD | %SystemRoot%\symbols\<binary ext>\ | %SystemRoot%\ |

**Note**

When debugging with Microsoft® Visual C++® development system on Windows NT/Windows 2000, do not use the WIN32API splitsym development tool. WIN32API splitsym copies private symbolic information from the <binary>.dbg file into the symbol directory under the binary extension (for example, %SystemRoot%\symbols\dll\ ), then deletes the original file from the binary directory. Visual C++ uses relies on finding private symbolic information (<binary>.dbg) file in the same directory as the binary itself, so for debug builds, it is necessary to only copy the private symbolic information and not to delete it. This is only an issue when using Visual C++ with Windows NT/Windows 2000. Consult Visual C++ documentation and Windows 2000 Driver Development Kit (DDK) documentation for further debugging information.

# Compiling DirectX Samples and Other DirectX Applications

[This is preliminary documentation and subject to change.]

This section provides information about considerations specific to compiling DirectX applications. The following topics are discussed:

- Preparing for Compilation
- Component Version Constants

# Preparing for Compilation

[This is preliminary documentation and subject to change.]

The samples included in this SDK use Microsoft® Visual C++® project files (MDP files) that describe the appropriate source files, project resources, and linker settings for each sample. However, you may need to make additional preparations to ensure that the samples compile and link properly, or you might need to prepare settings for a new project of your own. (For the samples, developers that use other compilers can reference the generic makefiles included for information about the required libraries, then configure their environment appropriately.) The information provided here applies to the DirectX samples and the DirectX applications you will create.

After opening a project file in Visual C++, you should verify some settings before trying to compile the application. The following descriptions are given in terms of the Microsoft Visual C++ 5.0 and 4.2 options. Previous versions of Microsoft Visual C++ and other compilers have equivalent settings. If you do not use these versions of Microsoft Visual C++, refer to the documentation provided with your development environment for information on changing these settings.

### Note

The following discussion uses the default installation paths (C:\mssdk\include and C:\mssdk\lib) to describe file locations. Your installation paths might differ.

## Include search paths

Make sure the search path for header files is correct, and that the directory for DirectX header files is the first path that the compiler searches. To check the include path, choose **Options** from the **Tools** menu, then select the **Directories** tab. The following dialog box will appear.

The topmost path should indicate the folder that contains the latest DirectX header files. The default path is C:\mssdk\include. If the path is not present, add it to the list and move it to the top of the search list by using the toolbar controls within the **Directories** tab.

## Linker search paths

Check the search paths and search order that the linker uses to search for link libraries. The link search paths are also listed on the Directories tab; choose **Options** from the **Tools** menu, then select the **Directories** tab. When the dialog appears,

choose the "Library files" option in the "Show directories for:" drop-down list box. The topmost path should be the folder that contains the latest DirectX link libraries. The default path is C:\mssdk\lib. (Borland link libraries are provided in the Borland folder within the default folder.)

### Project link libraries

If you are using the provided sample project files, you shouldn't need to verify these settings, as they are specified with the provided project files. For new applications, choose **Settings** from the **Project** menu to make the following dialog box appear. (In Visual C++ 4.2, this is the **Settings** item on the **Build** menu.)

All DirectX applications should link to the Dxguid.lib include library, which defines the globally unique identifiers (GUIDs) for all DirectX foundation COM interfaces. (Alternatively, you can define INITGUID prior to all other include and define statements in a single source module.) In addition, verify that the application is linked to the appropriate standard DirectX link libraries. The following table shows the various DirectX foundation components that might be used, and the corresponding link libraries for those components.

| **Component** | **Link Library (*.LIB)** |
|---|---|
| DirectDraw | Ddraw.lib |
| Direct3D Immediate Mode | Ddraw.lib |
| DirectSound | Dsound.lib |
| DirectInput | Dinput.lib |
| DirectSetup | Dsetup.lib |
| (All components) | Dxguid.lib |

# Component Version Constants

[This is preliminary documentation and subject to change.]

For backward compatibility with previous versions of DirectX, some DirectX components include variable API element definitions in their header files. Affected elements are typically capability structures or flag sets that are version specific. Parts of some header files are surrounded by preprocessor conditionals that cause the preprocessor to effectively "filter-out" unneeded definitions. The value of the defined constant identifies a specific version of the component; if no value is defined, the headers set a value that identifies the DirectX version for which the header file was written. An example from the DirectDraw header file, ddraw.h, is shown here:

```
#ifndef   DIRECTDRAW_VERSION
#define   DIRECTDRAW_VERSION 0x0600
#endif /* DIRECTDRAW_VERSION */
```

The following table includes the components that use these version constants, their labels, and their default value for this release of DirectX.

| Component | Label | Value |
|---|---|---|
| DirectDraw | DIRECTDRAW_VERSION | 0x0600 |
| Direct3D Immediate Mode | DIRECT3D_VERSION | 0x0600 |
| DirectInput | DIRECTINPUT_VERSION | 0x0600 |

You can define other values for these constants to use newer versions of the header files with previous versions of the components. For example, to use the latest headers to compile against the DirectX 3.0 version of DirectDraw, you would define DIRECTDRAW_VERSION to be 0x0300. You can set the value for the constant in your development environment, or you can change the value in the header file itself.