9.8 ActiveX: Handling Events

WHAT YOU LEAI

The ActiveX control provides an abstracted list of events to handle. These events do not map directly to the events generated by the filter or the filter graph manager, but they are important events at such a high-level interface. For example, you receive events when the state of the movie changes or when the position of the movie changes.

To handle such events, you can use the Microsoft Visual C++ class wizard to add a handler for each event. Notice that you have to select the DirectShow ActiveX control ID that you specified in the resource editor in order to display its events. Figure 9-7 shows how to add a handler for the *StateChange* event.

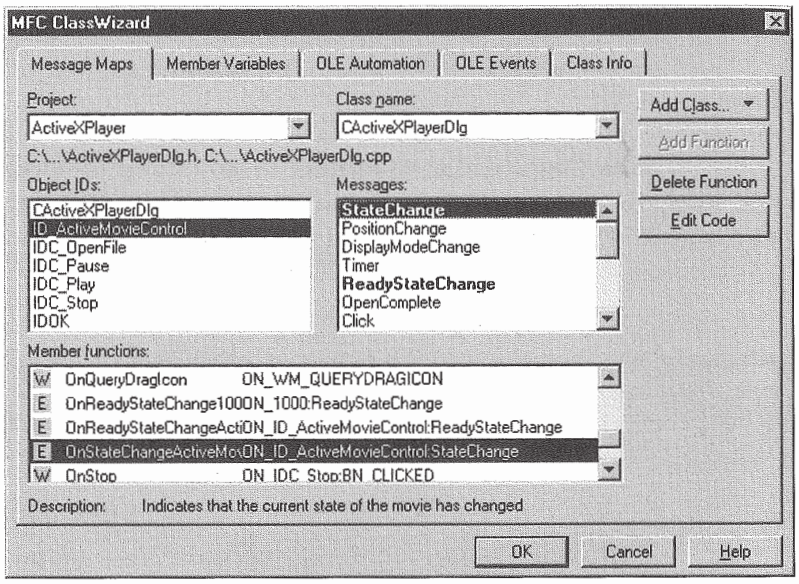


FIGURE 9-7 Handling DirectShow ActiveX control events using the Microsoft Visual C++ class wizard.

WHAT HAVE YOU LEARNED?

By the end of the chapter, you should have learned how to build DirectShow filter graphs from within your application in one of three ways: with the ActiveX controls, the automatic COM interface, or the manual COM interface.

You should also know how to access your filter's custom interface and its property pages. You should be able to handle filter events and control the running state of the DirectShow filter.

Lastly, you should be able to enumerate all the registered DirectShow filters, loaded filters, and pins in a filter.

CHAPTER 10

Mixing Sprites, Backgrounds, and Videos

PART III

WHY READ THIS CHAPTER?

It was certainly nice to play a video clip with DirectShow, but wouldn't it be even better if you could use video as part of your game or application? Surely, video is not the only thing moving on the screen; you probably have some moving sprites, backgrounds, and animation bouncing around the screen at the same time.

In this chapter we'll show you how to

- mix multiple objects together and how to place them relative to each other—in the front, middle, or the back of the viewing area, and
- use RDX to mix video, backgrounds, and sprites.

10.1 Introduction to Mixing

You've seen how mixing works throughout Part II when we showed you how to superimpose a static sprite on top of a static background using GDI, DirectDraw, and RDX. That was nice, but it can get boring fast.

Typically, multimedia applications have multiple sprites, backgrounds, animation, and video clips moving around on the screen all at the same time—sometimes with music playing in the background. For example, you could play a video clip with animation moving in the front and a moving background.

10.1.1 Mixing Sprites with Video

First let's review how we draw a static sprite on top of a static background. As you recall, some of the pixels in the sprite are transparent and should not be drawn on top of the background. For optimal performance, we typically mix the two objects in system memory and then write the mixed result to the screen.

As you can see in Figure 10-1, you can first copy the background to the mixing buffer, then overlay the sprite on top of it. Actually, you can overlay many sprites on top of the background at this stage. Finally, you can write the mixed result to video memory to make it visible on the screen. Notice that you only have to update the display whenever the sprite or the background moves around the screen.

With motion video, you display so many frames per second (fps) to give the illusion of motion. Now, if you treat every frame in the video as if it were a static background, you can apply the same technique we just discussed, mixing sprites with a background, for mixing sprites over video. In this case, however, you need to update the display screen whenever the sprite or the video moves on the screen *and* whenever a new video frame is displayed.

10.2 I

10.1.2 Mixing Animation with Video

Suppose you want to mix an animation sequence on top of video—an animation clip is a sequence of sprites with transparent pixels, which gives the effect of a moving picture. Similar to motion video, animation clips are displayed at a specific rate measured in frames per second. To mix an animation on top of a video, you can use the same concept as when you mix a sprite over video.

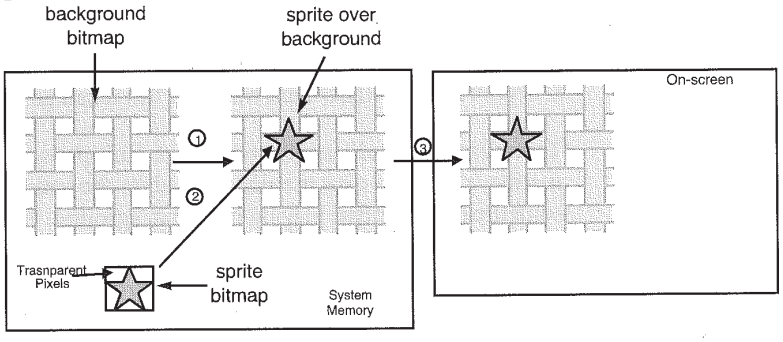


FIGURE 10-1 Mixing a static sprite with a static background.

At any given moment, you only need to deal with one sprite from the animation and one frame from the video. This is exactly the case when we displayed a sprite over a static background. In this case, however, you need to update the display whenever a new frame is displayed from either the animation or the video *and* whenever the animation or the video hops around the screen.

Of course, you can apply this same technique for mixing a video on top of another video. The same technique is used on TV shows and in the movies when there is a scene inside a car and the back window shows some video clip giving the illusion that the car is moving. To do that, you typically film the car in front of a blue background—blue is your transparency color. Then you mix this video clip, with the blue background, with another video clip exactly the same way you mixed animation over video.

10.2 Mixing with RDX

In Part II you've learned how to use RDX for mixing static sprites on top of static backgrounds. Here we'll show you how to use RDX to mix a static sprite over video. You can use the same technique to superimpose video over video or animation over video.

Before we go into that, let's first review some of the techniques RDX uses to perform object mixing. RDX uses a *draw order* to decide which object should be rendered first on the screen. For example, if you want to give the illusion that a background is "behind" a sprite, you would assign the background a higher draw order number than the sprite. RDX in turn paints the background first, then overlays the sprite on top of it (Figure 10-2).

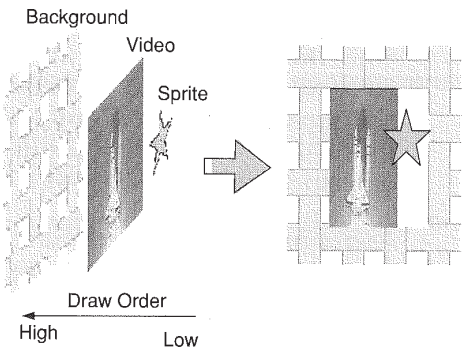


FIGURE 10-2 RDX draw order. Higher order objects are displayed behind lower order objects.

10.2.1 Playing Video with the RDX DirectShow Interface

First let's show you how to play a video clip using RDX. RDX supports multiple architectures for video playback, such as DirectShow and VFW. Since we've been discussing DirectShow, we will show you how to play a video clip using that interface. In this example, we'll use an MPEG file as the video clip.

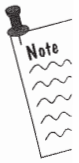
To display an MPEG file, we first ask RDX to create a filter graph object and associate the MPEG file with it. RDX in turn creates a DirectShow filter graph for the input file and returns a handle to an RDX video object that represents that file.

```
fgCreate(&m_hAM);
fgAssociateFile(m_hAM, "blastoff.mpg");
fgGetVideoObject(m_hAM, 0, &m_hVid);
```

Create a DirectShow filter graph "fg" object and set the MPEG file as the source. If successful, get a pointer to the RDX video object for later use.

Now you can instruct DirectShow to place the final output to the RDX surface, *hSurf* (refer to Chapter 6 to learn how to create an RDX surface). You can then set the draw order for the video such that it would be drawn behind the sprite. As an example, we use 100 for the sprite and 150 for the video clip. Finally, we declare that this object is visible.

```
objSetDestination (m_hAM, hSurf);
objSetDrawOrder (m_hAM, 150);
objSetVisibility(m_hAM, TRUE);
```



When DirectShow renders the MPEG file, it writes the final image not to the screen but to the RDX surface, which is typically in system memory or offscreen video memory. To display each frame to the screen, you must call the *srfDraw()* function, which copies the contents of the RDX surface to the appropriate location on the screen.

Alternatively, you can request that RDX automatically call the *srfDraw()* function to render each frame to the screen. To do that, you can use RDX's timers and events to schedule a draw event every frame. A *timer* is an object that counts user-defined time periods. An *event* is an object that defines an activity that you want to perform on an episodic or periodic basis.

To create a timer, you must call the *TimerCreate()* function with a handle to the video object and the timer sampling rate. You can then activate the timer with the *TimerStart()* function. Even though the timer is generating so many ticks per second, the timer tick does not generate any callback or event. So what use is this timer anyway ?

```
fgGetVidInfo(m_hAM, m_hVid, FG_INFO_SAMPLERATE, &dwFPS);
timerCreate((WORD)dwFPS, (HTIMER *)phTimer);
timerStart(*phTimer);
m_hTimer = *phTimer;
```

CAUTION: Make sure to stop *phTimer* before you destroy the RDX objects. Use *TimerStop()* or *TimerDestroy()*.

To make it worthwhile, you must associate the timer with a draw event—a draw event informs RDX to call the *srfDraw()* function. When the timer ticks, it sends RDX a draw event advising it that it is time to render the frame. To create a draw event, you must call the *eventCreate()* function with `EVENT_DRAW` as a parameter. You must then associate the event with the timer that will raise that event.

```
if (bAutoDraw) {
    eventCreate(m_hObj, EVENT_DRAW, 0, 0, &m_hDrawEvent);
    eventSchedule(m_hTimer, m_hDrawEvent, 1, RELATIVE_TIME, 1, 0xffff);
}
```

The third paramater, wPeriod, in *eventSchedule* allows you to specify the number of timer ticks per event period, for example, if the timer generates 30 ticks/sec and the event wPeriod is 3, then an event is generated every 3 timer ticks.

Even though DirectShow decodes the video clip according to its frame rate, the number of frames rendered to the screen depends on the timer's sampling rate and the event's period.

Now that you have everything set up, you can call the *fgPlay()* function to put the DirectShow filter graph in run state.

```
objPrepare(m_hAM);
fgPlay(m_hAM, PLAYMODE_REPEAT, 0, 0);
```

At this stage, DirectShow decodes every frame into an RDX surface, the timer generates a draw event on every decoded frame, and RDX calls the *srfDraw()* function to copy the image from the RDX surface to the screen.

Why don't you fire up the sample application on the CD and select the option for this chapter from the menu. You should see a video clip playing on the screen.

10.2.2 Mixing a Sprite on Top of Video

Now that you have the video playing, let's see how we can overlay a sprite on top of it. As in Chapter 6, you must first load the sprite bitmap into memory and create an RDX sprite object. Once the sprite object is created, you can associate it with the RDX surface, *hSurf*.

```
bitmap bm;
bitmap.GetBitmap(&bm);
m_dwWidth = bm.bmWidth;
m_dwHeight = bm.bmHeight;
m_byTransp = byKeyColor;
```

Create and set up an hbmp (Source Data Object).

```
HBMPHEADER bmpHeader;
hbmpCreate(m_dwWidth,m_dwHeight,RGB_CLUT8,&m_hBmp);
BYTE *pData;
hbmpGetLockedBuffer(m_hBmp, &pData, &bmpHeader);
bitmap.GetBitmapBits(m_dwWidth*m_dwHeight, pData);
hbmpReleaseBuffer(m_hBmp);
hbmpSetTransparencyColor(m_hBmp, (DWORD)byKeyColor);
```

Create sprite; associate data to it; associate sprite to surface.

```
sprCreate(&m_hSpr);
sprSetData(m_hSpr, m_hBmp);
objSetDestination(m_hSpr, m_hSurf);
```

Finally, you need to set the draw order and visibility of the sprite. Notice that we set the draw order to be lower than that of the video clip so that the sprite is drawn in front of the video clip.

```
objSetDrawOrder(m_hSpr, 100);
objSetVisibility(m_hSpr, TRUE);
```

WHA
YOU LE

10.2.3 Mixing Video on Video

As we've mentioned earlier, you should be able to mix an animation or a video on top of another video. Let's see how you can overlay a video clip with a transparency color on top of another video.

You can actually use the same code from section 10.2.1 to start the video clip in the foreground—with a couple of modifications. First, you must inform RDX about the transparent color of the video. To do that, you must call the *fgvidSetTransparencyColor()* function.

```
fgvidSetTransparencyColor(m_hAM2, (DWORD)byKeyColor);
```

As with the sprite, you should set the draw order of the video clip to be in front of the background video clip. Notice that in our example we positioned this video clip between the background video (150) clip and the sprite (100).

```
objSetDrawOrder (m_hAM2, 120);
```

At this stage, you'll have two video clips playing, one on top of another with a sprite in front of both of them. Notice that since the two video clips could have different frame rates, you need to use the higher frame rate when you create the timer for the draw event. This way, you're drawing at the rate of the faster video clip.

WHAT HAVE YOU LEARNED?

By now you should be familiar with mixing different objects on top of each other. In this chapter you learned

- about mixing sprites, video, and animation together,
- about draw order and how to position objects relative to each other,
- about RDX timers and events and how to create them,
- how to use RDX to mix a sprite on top of video, and
- how to mix video or animation on top of another video.

CHAPTER 11

Streaming Down the Superhighway with RealMedia

WHY READ THIS CHAPTER?

The Internet! You must have heard of it by now. Yes, and while cruising the Net, you must have been struck by all of these RealAudio icons *"To Listen, Click Here*  *."* RealAudio has become THE audio streaming solution on the Internet.

With its success, RealNetworks released a similar technology for video on the Internet—RealVideo. In 1997 the company is building on its success with streaming on the Internet and is releasing a new streaming architecture, which allows for installable media types to be streamed on the Internet. This technology is called RealMedia.

In this chapter, you will

- get an overview of RealMedia and learn about its plug-in model,
- be introduced to the concept of a RealMedia plug-in and how to build File-Format and Rendering plug-ins,
- learn about Audio Services and how to use them within a plug-in, and
- learn about metafiles and how to use them.

In the past few years, the number of people connected to the Internet has grown astronomically. Similar to television, radio, and newspaper, the Internet has become the information medium of choice for millions of people. The Internet, however, offers an additional quality that does not exist in any of the earlier mediums: interactivity. Televisions and radios allow you to select between a preset number of local or cable channels; the Internet, on

the other hand, opens the gate to millions of information servers, games, and music archives throughout the world.

RealNetworks (RN) seized the opportunity and established itself as THE Internet audio streaming technology on the Internet. Its RealAudio technology is specifically designed for real-time audio streaming on the Internet. With real-time audio streaming, you don't have to download an audio file first and then play it back; rather, you play the data as you retrieve it from the Internet server. Building on their success with RealAudio, RN introduced a similar streaming technology for video called RealVideo, and then RealMedia.

RealMedia is a real-time streaming technology specifically designed for the Internet. RealMedia includes both the RealVideo and RealAudio technologies as part of its core. With the plug-in mechanism that it provides, you can stream and synchronize the playback of any data type, in real time, over the Internet. For example, you can stream a new file format like MPEG, text, animation, MIDI, financial data, weather information, industrial information, or VRML.[1]

In our effort to present only technologies of the future, we wrote this chapter while the RealMedia SDK was still in its late beta cycle. Therefore some of the APIs might have changed slightly by the time this book is published. Nonetheless, the material in this chapter should be relevant and reflect the RealMedia architecture accurately. Use this chapter for the concept, but use the RealMedia SDK for the actual API definitions.

11.1 Overview of RealMedia

RealMedia is an open, cross-platform technology for streaming multimedia presentations over the Internet—or networks in general. (See Figure 11-1.) It uses the Real Time Streaming Protocol (RTSP) for communicating over the Internet[2] and the Real Time Session Language (RTSL) to define presentations. What does all of this mean?

11.2

1. VRML: Virtual Reality Modeling Language.
2. RTSP supports multicasting, unicasting, and RTP protocols.

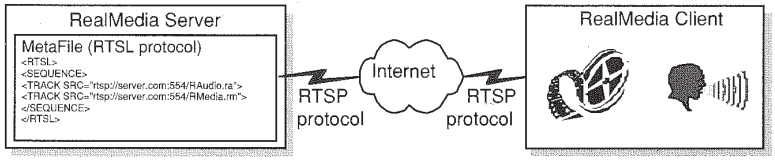


FIGURE 11-1 The roles of the RTSP protocol and RTSL session language.

RealMedia uses the RTSP protocol to transport data across networks—both the Internet and Intranets.[1] RTSP defines an application interface for controlling the delivery of real-time data. It allows for delivery of multiple streams simultaneously, such as video and audio and time-stamped data packets. For a reliable delivery, the RealMedia client uses the RTSP protocol to acknowledge the server when a packet is received; otherwise, the client resends another request for the packet or decides to throw it away. This decision depends on the quality setting of the application.

RTSL is a presentation language that is similar in form to the HyperText Markup Language (HTML). HTML is used to create Web documents on the Internet. RTSL allows you to define a presentation sequence that consists of multiple audio, video, and other data streams. With RTSL, the RealMedia server and client can negotiate the type of content delivered based on the information in the RTSL file (a.k.a. a metafile) and the settings of the player. For example, in the metafile, you can specify different media files (audio, video) depending on the bandwidth of the Internet pipe (28.8K, ISDN, and so forth) and on the language (English, French, and so forth). For a 28.8K pipe you can deliver a file with low quality and a low rate of data; for ISDN, you can deliver a better quality file with a higher rate of data. (See the RTSL definition in the RealMedia SDK for more details.)

We won't go into all the details of RTSP and RTSL in this book. Since RealMedia handles all the communication between the client and the server internally, you never have to deal directly with the RTSP protocol. However, as a content developer (someone who designs metafiles), you will need to learn more about the RTSL protocol and how to use it. Refer to RealMedia SDK for more details.

11.2 The RealMedia Plug-in Architecture

RealMedia is a simple plug-in architecture for adding custom data types. Figure 11-2 shows three RealMedia plug-in interfaces: *File-System, File-Format,* and *Rendering* plug-ins.

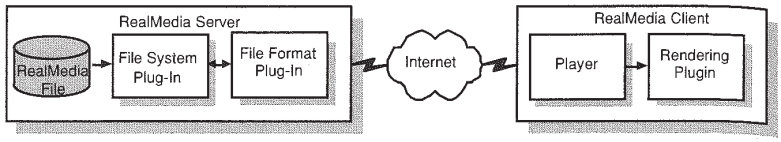


FIGURE 11-2 RealMedia plug-in architecture.

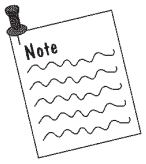
The *File-System* plug-in is only responsible for reading "raw" data from a source. The source could be a prerecorded audio/video file, a satellite feed, or a database server. This plug-in is typically loaded by the RealMedia server. The File-System plug-in does not know, or care, how the data will be parsed; it only knows how to read, write, and seek through a file. Since the RealMedia binaries come with a slew of File-System plug-ins, you typically don't have to implement a File-System plug-in to stream custom data types.

Supported Datatypes

Text	*Text*
AVI	*Audio/video*
MOV	*QuickTime*
WAV	*Audio*
SND	*Audio*
AIFF	*Audio*
AU	*Audio*

The *File-Format* plug-in is responsible for parsing the data, splitting it into multiple streams, and breaking it into smaller packets for delivery over the Internet. This plug-in is typically loaded by the RealMedia server. The File-Format plug-in does not know how to read the data from the source, and it does not know how to send the data over the Internet. Currently, RealMedia supports AVI, WAV, AU, SND, AIFF, RealAudio, RealVideo, RealMedia, and RealText file formats.

The *Rendering* plug-in understands the contents of the data and knows how to render it to its final destination—screen, audio device, and so forth. This plug-in is typically loaded by the RealMedia player or client.



In Figure 11-2, we show that the File-System and File-Format plug-ins are loaded by the RealMedia server. If you're playing a RealMedia file on a local machine, the RealMedia player loads the File-System, File-Format, and the Rendering plug-ins on the same PC.

Notice that none of the plug-ins we've discussed so far deal with data delivery over the Internet. They only worry about reading the data, breaking it into smaller packets, and rendering the final result. The RealMedia server and client handle all the necessary communication over the Internet. RealMedia allows for streaming any data type and synchronizing the playback of multiple data types.

So what do you really need to do to stream your own custom data type? Typically, you only need to implement a File-Format plug-in and a Rendering

plug-in, since they both have to understand the new data type. The File-System plug-in, on the other hand, is only required if you have to read data from a source not supported by the RealMedia binaries, for example, from a database server.

In this chapter, you'll learn how to build a File-Format and a Rendering plug-in. You'll also learn about RealMedia metafiles and how to use them to configure the Web server.

Let's go over some of the basic RealMedia concepts and interfaces. First we'll describe the data flow model between the server and client. Then we'll glance over some of the basic RealMedia interfaces that are used in the sample code in this chapter.

11.3 Data Flows: Server to Client

For the purposes of this discussion, we're assuming that you know how to use a Web browser such as Internet Explorer or Netscape Navigator. When you select a hot link in the browser, it takes you to a new Web page or downloads a file to your local drive. If the hot link points to an audio or a video file, the browser first downloads the file to your local machine and then launches the media player to play it. Web browsers allow you to associate any file extension with an application that will be launched when such a file is downloaded. For example, **.doc* is associated with launching *WinWord*.

To perform realtime streaming, RealMedia adds another step to this process. Instead of pointing the hot link to the RealMedia file on the server, you point it to a *metafile*. Metafiles hold configuration information that allows the client (RealPlayer) to communicate directly with the server. They also hold the list of media files to play when the metafile hot link is selected. We'll discuss metafiles in more detail later in this chapter.

So what really happens when you select a metafile hot link? Since the metafile file extension **.rts* is associated with the RealMedia player, the player is launched when the metafile hot link is activated. The player parses the metafile to find the streams that it should request from the RealMedia server. Notice that once the metafile is downloaded, the player makes the connection directly to the RealMedia server and bypasses both the Web browser and the Web server.

On the server side, the server loads the appropriate plug-ins and starts delivering data packets to the player (Figure 11-2). The RealMedia server

loads a File-System plug-in to read the raw data from a file. It then loads the appropriate File-Format plug-in based on the file extension of the media file. The File-Format plug-in parses the media file and determines the MIME type of each stream in the file.[3] The server sends the MIME type of each stream to the client, over the Internet, and the RealMedia client, in turn, loads the appropriate Rendering plug-in for that MIME type.

Once the plug-ins are loaded, the RealMedia server requests a data packet from the File-Format plug-in. The File-System plug-in reads the raw data from the file, the File-Format plug-in parses it and breaks it into smaller packets. The RealMedia server sends the packet over the Internet to the client where it is rendered by the Rendering plug-in.

In a nutshell, File-Format plug-ins make the packets, Rendering plug-ins receive the packets and play them, and the RealMedia engine handles all the underlying communication and timing of shuttling the packets from the server to the player.

11.4 Data Management Objects

RealMedia defines a set of data objects to transport the data from the server to the client. These objects include dynamic memory allocation, indexed lists, and data packet objects.

Although all the RealMedia objects are COM interfaces, they are not specific to the Windows environment. Even though COM was defined for Windows, the COM architecture does not require Windows.

11.4.1 IRMABuffer: Dynamic Memory Allocation Object

The IRMA[4] Buffer object allows you to allocate a memory buffer at runtime. Typically, the buffer is used to transport data over the Internet. To allocate a memory buffer in RealMedia, you need to create an instance of the IRMA-Buffer object, set the size of the buffer, and then request a pointer to it. And you thought *malloc()* was hard to use!

3. A MIME type specifies the type of data in the message. MIME, or Multipurpose Internet Mail Extension, allows for transporting mail messages with binary data and many parts such as attachments and such.
4. IRMA: Interface RealMedia Architecture.

To create an instance of the IRMABuffer object, you need to call the *IRMACommonClassFactory::CreateInstance()* function using `CLSID_IRMABUFFER` as a parameter. You'll soon learn how to request a pointer to an IRMACommonClassFactory object. To set the size of the buffer, you must call the *IRMABuffer::SetSize()* member function; the actual memory allocation happens here. If successful, you can then call the *IRMABuffer::GetBuffer()* function to obtain a pointer to the data buffer. When you're done with the buffer, you should release the object to avoid any memory leaks.

```
m_pClassFactory->CreateInstance(CLSID_IRMABuffer,(void**)&pTitle);
pTitle->SetSize(INFO_SIZE+1);
pTitleData= (char*)pTitle->GetBuffer();
strncpy(pTitleData,pBufferData,INFO_SIZE);
pTitle->Release();
```

IRMABuffer functions:
Get()
Set()
SetSize()
GetSize()
GetBuffer()

11.4.2 IRMAValues: Indexed List Object

The IRMAValues object allows you to build an indexed list at runtime and send it off to other plug-ins over the Internet. The index is an ASCII string that specifies some special property. The value is either an IRMABuffer object or an unsigned long. For instance, you could use the IRMAValues object to build the following indexed list.

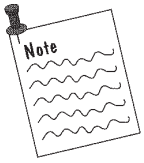
Index	"Title"	"Author"	"Copyright"	"Count"
Value	*"Carrots"*	*"Bugs Bunny"*	*"BigEars Inc"*	*3*

As with the IRMABuffer object, you must first create an instance of the IRMAValues object with the *IRMACommonClassFactory::CreateInstance()* function. You can then call the member function *SetPropertyULONG32()* or *SetPropertyBuffer()* to add an unsigned long or an IRMABuffer object to the list, respectively. Notice that the string index, for example, "title," is specified in the first parameter.

```
m_pClassFactory->CreateInstance(CLSID_IRMAValues, (void**)&pHeader))
pHeader->SetPropertyBuffer ("Title", pTitle);
pHeader->SetPropertyULONG32("StreamCount", 1);

pHeader->GetPropertyBuffer("Title", pTitle);
pTitleData = pTitle->GetBuffer();
pTitle->Release();
pHeader->Release();
```

IRMAValues functions:
SetPropertyULONG32 ()
GetPropertyULONG32()
GetFirstPropertyULONG32()
GetNextPropertyULONG32()
SetPropertyBuffer()
GetPropertyBuffer()
GetFirstPropertyBuffer()
GetNextPropertyBuffer()

To retrieve an item from the list, you can call the *GetPropertyBuffer()* or the *GetPropertyULONG32()* function for an IRMABuffer object or an unsigned long, respectively. In addition, you can enumerate the entire indexed list with the *GetFirstXyz()* and *GetNextXyz()* member functions. Refer to the Include file in the RealMedia SDK for prototypes of these functions.



11.5

The following rules describe when to use the *AddRef()* and *Release()* functions with RealMedia objects:

- If an object is passed to your code as a parameter of a function call, you must use *AddRef()* to reference the object. When your code is finished with the object, you must use *Release()* to release the object.

```
Void Function (IRMAObject  *pObject) {
    pObject->AddRef();
    ...Use Object Here...
    PObject->Release();
```

- For objects returned by functions, use *AddRef()* to reference the object inside the function. You must use *Release()* to release the object when you're done with it. The following functions use the *AddRef()* function to increment the reference count of the objects before returning them:

  RMACreateInstance

  IRMAFileSystem::CreateFile

  IRMACommonClassFactory::CreateInstance

  IUnknown::QueryInterface

  IRMAFileSystem::CreateDir

- If your code creates an object using the C++ new operator, your code must use the *AddRef()* function to reference the object. When your code is finished with the object, it must use *Release()* to release the object.

11.4.3 IRMAPacket: Packet Transport Object

The IRMAPacket *object* is used to transport data packets from the File-Format plug-in on the server side to the Rendering Plug-in on the client side.

Again, you must first call the *IRMACommonClassFactory::CreateInstance()* function to create an instance of the IRMAPacket object. You can then call the *Set()* member function to specify the IRMABuffer object that holds the data of each packet. With the *Set()* function you can also set a time stamp for the packet and a priority flag indicating the importance of the packet—Can it be dropped or not?

```
m_pClassFactory->CreateInstance(CLSID_IRMAPacket,(void**)&pPacket))
pPacket->Set(pBuffer,m_ulCurrentTime,0,0,PN_RELIABLE_NORMAL);
pPacket->Release();
```

11.5 RealMedia Asynchronous Interfaces

File-Format plug-ins are responsible for parsing the data and splitting it into multiple streams. They're also responsible for breaking the data in each stream into smaller packets before sending it over the Internet.

In addition to the IRMAPlugin interface, File-Format plug-ins implement both the **IRMAFileFormatObject** and **IRMAFileResponse** interfaces. The IRMAFileFormatObject interface defines the functionality of the DLL as a File Format plug-in. The RealMedia server uses this interface to retrieve header information from the source file and the header for each stream. It also uses this interface to request data packets to send out over the Internet.

IRMAFileResponse is a callback interface used to notify the plug-in when an asynchronous operation is complete. As you recall, the File-Format plug-in uses the services of the File-System plug-in to read raw data from the input source. Since all the File-System plug-in's operations are asynchronous, the File-Format plug-in exposes the IRMAFileResponse interface in order to receive notification when these operations are complete.

RealMedia defines nonblocking interfaces for the File-Format and the File-System plug-ins. These asynchronous interfaces allow the server to process requests from the clients while the plug-ins are busy preparing data packets from the input source.

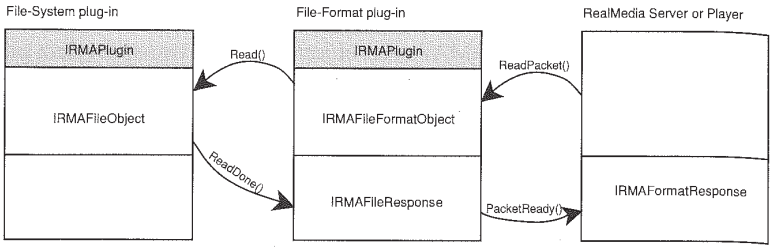


FIGURE 11-3 RealMedia asynchronous interfaces.

Suppose that the RealMedia server wants to get a data packet to transmit over the Internet. The server calls the File-Format plug-in function *IRMAFileFormatObject::GetPacket()* to obtain a data packet. The *GetPacket()* function, in turn, calls the File-System function *IRMAFileObject::Read()* to read a block of raw data from the input source. Now, since both the IRMAFileFormatObject and the IRMAFileObject interfaces are asynchronous, both the *Read()* and the *GetPacket()* functions return before the packet is created. At this stage, the server could process previous packets in the queue.

When the raw data is read from the input source, the File-System plug-in calls the File-Format plug-in function *IRMAFileResponse::ReadDone()* to notify it that the raw data read is done. The File-Format plug-in, in turn, calls the server back at the *IRMAFormatResponse::PacketReady()* function to notify it that the packet is ready to send.

11.6 Common Requirements for All Plug-ins

Although we're only dealing with Windows 95 in this book, it's worth mentioning that RealMedia is a cross-platform architecture. Nonetheless, writing a File-Format or a Rendering plug-in should require little or no operating system services. The File-System plug-in, on the other hand, is platform dependent since it requires direct access to the input devices—file, satellite, and so forth.

To create RealMedia plug-in, you must keep the following in mind:

- The plug-in you create must export the C-style function *RMACreateInstance()*, which is used to create an instance of the plug-in. This function must be exported externally in the DEF file.
- The plug-in must implement the base IRMAPlugin interface. RealMedia applications use this interface to retrieve information about the plug-in.

- Each plug-in must implement at least one additional plug-in object that defines the functionality of the plug-in. For example, Rendering plug-ins implement the IRMARenderer object, and File-Format plug-ins implement the IRMAFileFormatObject.

When a RealMedia application loads a plug-in, it calls the *RMACreateInstance()* function, which creates an instance of that plug-in and then returns a pointer to it.

```
STDAPI RMACreateInstance(IUnknown** ppIUnknown)
{
    *ppIUnknown = (IUnknown*)(IRMAPlugin*)new CExampleFileFormat();
    if (*ppIUnknown) {
        (*ppIUnknown)->AddRef();
        return S_OK;
    }
    return E_OUTOFMEMORY;
}
```

Make sure to Export this function in the DEF file.

Once the plug-in is created, RealMedia calls the *IRMAPlugin::InitPlugin()* function to initialize it. This function accepts only one parameter, PCONTEXT, which allows you to retrieve a pointer to the common class factory. Remember, this is the IRMACommonClassFactory object we used to create the buffers earlier in the chapter.

```
STDMETHODIMP CExampleFileFormat::InitPlugin(IUnknown* pContext)
{
    m_pContext = pContext;
    m_pContext->AddRef();
    m_pContext->QueryInterface(IID_IRMACommonClassFactory
            (void**)&m_pClassFactory);
    return S_OK;
}
```

IRMAPlugin functions:
InitPlugin()
GetPlugininfo()

IRMACommonClassFactory provides a *CreateInstance()* function, which serves the same purpose as the COM *CoCreateInstance()* function.

The *IRMAPlugin::GetPluginInfo()* function is then called to retrieve filter-specific configuration information. The BLOADMULTIPLE parameter specifies whether multiple instances of this plug-in could be created at the same time—this must always be *true* for a File-Format plug-in. The other three parameters are self-explanatory.

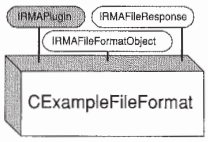
```
#define PCCHAR const char*
PCCHAR CExampleFileFormat::zm_pDescription    = "Example File Format Plugin";
PCCHAR CExampleFileFormat::zm_pCopyright      = "Your Company, All rights
reserved";
PCCHAR CExampleFileFormat::zm_pMoreInfoURL    = "http://www.yourcompany.com";

STDMETHODIMP CExampleFileFormat::GetPluginInfo (
    REF(BOOL)        bLoadMultiple,
    REF(const char*) pDescription,
    REF(const char*) pCopyright,
    REF(const char*) pMoreInfoURL
    )
{
    bLoadMultiple = TRUE;
    pDescription   = zm_pDescription;
    pCopyright     = zm_pCopyright;
    pMoreInfoURL   = zm_pMoreInfoURL;
    return S_OK;
}
```

REF(): parameters are passed by reference.

All plug-ins go through the same initialization up to this point. Other initialization is done based on the type of plug-in you're writing. Let's start with the File-Format plug-in and then move on to the Rendering plug-in.

11.7 Building a File-Format Plug-in

11.7.1 Initializing the File-Format Plug-in



Once the IRMAPlugin interface is initialized, the File-Format plug-in function *GetFileFormatInfo()* is called. This function returns a list of the supported MIME types along with the default file extensions of the supported input files. *This is it!* This is where you declare that this plug-in can support the list of file extensions.

```
#define PCCHAR const char*
PCCHAR CExampleFileFormat::zm_pFileMimeTypes[]  = {"application/x-yourfileformat", NULL};
PCCHAR CExampleFileFormat::zm_pFileExtensions[] = {"eff", NULL};
PCCHAR CExampleFileFormat::zm_pFileOpenNames[]  = {"Example File Format (*.eff)", NULL};
```

To support multiple streams and file formats, add the second, third, and so forth, stream information here. Make sure to terminate the list with `NULL`.

```
STDMETHODIMP CExampleFileFormat::GetFileFormatInfo (
    REF(const char**) pFileMimeTypes,
    REF(const char**) pFileExtensions,
    REF(const char**) pFileOpenNames
    )
{
    pFileMimeTypes  = zm_pFileMimeTypes;
    pFileExtensions = zm_pFileExtensions;
    pFileOpenNames  = zm_pFileOpenNames;
    return S_OK;
}
```

IRMAFileFormat-Object functions:
GetFileFormatInfo()
InitFileFormat()
GetFileHeader()
GetStreamHeader()
GetPacket()
Seek()
Close()

When the server receives a URL to load, it calls the *InitFileFormat()* function with that URL and two other object pointers: IRMAFileObject and IRMAFormatResponse. As we mentioned earlier, IRMAFileObject is an asynchronous interface used for reading, writing, or seeking a file. *IRMAFormatResponse* is used to notify the RealMedia server when an operation is complete.

In order for IRMAFileObject to notify us when an operation is complete, it needs to obtain a pointer to the File-Format plug-in's IRMAFileResponse interface. To do so, we must pass the pointer to "this" plug-in to the IRMAFileObject interface. IRMAFileObject, in turn, calls our *QueryInterface()* function to retrieve a pointer to the IRMAFileResponse interface.

```
STDMETHODIMP CExampleFileFormat::InitFileFormat(
    const char* pURL,
    IRMAFormatResponse* pFormatResponse,
    IRMAFileObject*  pFileObject
    )
{
```

Copy the URL to a member variable. Make sure to allocate enough space.

```
    if (m_pURL) {
        delete m_pURL;
        m_pURL = NULL;
    }
    if (pURL) {
        m_pURL = new char[strlen(pURL)+1];
        if (!m_pURL) return E_OUTOFMEMORY;
            strcpy(m_pURL,pURL);
    }
```

Save the File Object to read/seek data from the input source; save the File Response to notify the server when the operation is complete. We *AddRef()* these objects to keep them around for later use.

```
    m_pFFResponse = pFormatResponse;
    m_pFileObject = pFileObject;

    if (m_pFFResponse)    m_pFFResponse->AddRef();
    if (m_pFileObject)    m_pFileObject->AddRef();
```

Due to the asynchronous nature of the File Object, we use the variable m_state, to build a simple state machine to determine when an operations completes. Also, notice that we pass the file object a pointer to *this* plug-in so it could retrieve our callback interface, IRMAFileResponse.

```
    m_state = InitPending;
    m_pFileObject->Init(m_pURL, PN_FILE_READ | PN_FILE_BINARY, this);
    return S_OK;
}
```

Notice that you must expose the IRMAFileResponse interface by responding to the Global Unique Identifier (GUID), IID_IRMAFileResponse, in the *QueryInterface()* function.

```
STDMETHODIMP CExampleFileFormat::QueryInterface(REFIID riid, void** ppvObj)
{
    if (IsEqualIID(riid, IID_IRMAFileResponse)) {
        AddRef();
        *ppvObj = (IRMAFileResponse*)this;
        return S_OK;
    }

    *ppvObj = NULL;
    return E_NOINTERFACE;
}
```

Now, since the *IRMAFileObject::Init()* function is asynchronous, it returns before it initializes the object. Once the initialization is complete, IRMAFileObject calls the callback function *IRMAFileResponse::InitDone()* with the status of the operation. In turn, the *InitDone()* function calls the server's notification function *IRMAFormatResponse::InitDone()* to inform it that the File-Format plug-in initialization is complete. Remember that both the IRMAFileObject and IRMAFileFormatObject interfaces are asynchronous in nature.

```
STDMETHODIMP CExampleFileFormat::InitDone(PN_STATUS status)
{
    if (m_state != InitPending)
        return E_UNEXPECTED;

    m_state = Ready;
    m_pFFResponse->InitDone(status);
    return S_OK;
}
```

Pay special attention to the *m_state* member variable; we've used it to build a simple state machine to handle the asynchroneity of the IRMAFileObject interface.

11.7.2 File and Stream Headers

Once the plug-in is initialized and the file is ready for reading, the RealMedia server calls the *GetFileHeader()* function to retrieve the media file header. In this example, we seek to zero since the file header is located at the beginning of the file.

```
STDMETHODIMP CExampleFileFormat::GetFileHeader()
{
    if (m_state != Ready)
        return E_UNEXPECTED;

    m_state = GetFileHeaderSeekPending;
    m_pFileObject->Seek(0,FALSE);
    return S_OK;
}
```

Since the *Seek()* function is an asynchronous function, you have to wait for the *SeekDone()* callback before you can start reading from the file. When the seek is done, you can call the *IRMAFileObject::Read()* function to read the header from the file.

```
STDMETHODIMP CExampleFileFormat::SeekDone(PN_STATUS status)
{
    if (m_state == GetFileHeaderSeekPending) {
        m_state = GetFileHeaderReadPending;
        m_pFileObject->Read(FILE_HEADER_SIZE);
    }
}
```

Again, since the *Read()* function is an asynchronous function, the function *ReadDone()* will be called when the data is read. The *ReadDone()* function receives a status flag indicating the outcome of the read. If successful, the raw data is returned in the IRMABuffer object. You can then extract the data from the buffer and call the *IRMAFormatResponse::FileHeaderReady()* function to notify the server that the file header is ready.

```
STDMETHODIMP
CExampleFileFormat::ReadDone(PN_STATUS status, IRMABuffer* pBuffer)
{
    if (m_state == GetFileHeaderReadPending) {
        m_state = Ready;

        IRMAValues* pHeader;
        IRMABuffer* pTitle = NULL, *pAuthor = NULL;
        char       *pTitleData, *pAuthorData, *pBufferData;
```

Create an indexed list to hold the header, IRMAValues, and two buffers to hold the header data, pTitle & pAuthor. Set the buffer size appropriately and get a pointer to the buffer.

```
        m_pClassFactory->CreateInstance(CLSID_IRMAValues, (void**)&pHeader);
        m_pClassFactory->CreateInstance(CLSID_IRMABuffer,(void**)&pTitle);
        m_pClassFactory->CreateInstance(CLSID_IRMABuffer,(void**)&pAuthor);

        pTitle->SetSize(INFO_SIZE+1);
        pAuthor->SetSize(INFO_SIZE+1);

        pTitleData   = (char*)pTitle->GetBuffer();
        pAuthorData  = (char*)pAuthor->GetBuffer();
        pBufferData  = (char*)pBuffer->GetBuffer();

        strncpy(pTitleData,pBufferData,INFO_SIZE);
        pTitleData[INFO_SIZE] = '\0';

        strncpy(pAuthorData,pBufferData+INFO_SIZE,INFO_SIZE);
        pAuthorData[INFO_SIZE] = '\0';
```

Set the indexed list properties and inform the server that the file header is ready. Make sure to release whatever you've created to avoid memory leakage.

```
        pHeader->SetPropertyBuffer ("Title",       pTitle);
        pHeader->SetPropertyBuffer ("Author",      pAuthor);
        pHeader->SetPropertyULONG32("StreamCount", 1);

        m_pFFResponse->FileHeaderReady(status, pHeader);

        pHeader->Release();
        pTitle->Release();
        pAuthor->Release();
    }
}
```

Speicify the number of streams in the file here.

Notice that we specified the number of streams in the input file in the StreamCount index. For each stream, the server calls the *GetStream-Header()* function to retrieve the specific header for the stream. As when we were working with the file header, you would look for the stream header—asynchronously—and read the data at that location.

```
STDMETHODIMP CExampleFileFormat::GetStreamHeader(UINT16 unStreamNumber)
{
    // Seek to the stream header
    if (m_state == Ready) {
        m_state = GetStreamHeaderSeekPending;
        m_pFileObject->Seek(FILE_HEADER_OFFSET+FILE_HEADER_SIZE,FALSE);
    }
}

STDMETHODIMP CExampleFileFormat::SeekDone(PN_STATUS status)
{
    // read the stream header
    if (m_state == GetStreamHeaderSeekPending){
        m_state = GetStreamHeaderReadPending;
        m_pFileObject->Read(STREAM_HEADER_SIZE);
    }
}
```

When the read is done, the *ReadDone()* function is called with the data in the IRMABuffer object. As with the file header, you can retrieve the data from the buffer and pass it on as an indexed list.


```
STDMETHODIMP
CExampleFileFormat::ReadDone(PN_STATUS status, IRMABuffer* pBuffer)
{
    if (m_state == GetStreamHeaderReadPending) {
        m_state = Ready;

        IRMAValues* pHeader;
        m_pClassFactory->CreateInstance(CLSID_IRMAValues,(void**)&pHeader);
```

This is where you specify the output type of the stream; this MIME type also specifies what kind of rendering plug-in must be loaded to render this stream.

```
        IRMABuffer*     pMimeType = NULL;
        char    szMimeType[] = "application/x-yourRenderFormat";
        m_pClassFactory->CreateInstance(CLSID_IRMABuffer,(void**)&pMimeType;
        pMimeType->Set((const UCHAR*)szMimeType,strlen(szMimeType)+1);
```

This is the stream header information. It includes the data buffer, pBuffer, bit rate, packet size information, timing, and stream type.

```
        pHeader->SetPropertyBuffer ("OpaqueData",     pBuffer);
        pHeader->SetPropertyULONG32("StreamNumber",   0);
        pHeader->SetPropertyULONG32("MaxBitRate",     MAX_BITRATE);
        pHeader->SetPropertyULONG32("AvgBitRate",     AVG_BITRATE);
        pHeader->SetPropertyULONG32("MaxPacketSize", MAX_PACKETSIZE);
        pHeader->SetPropertyULONG32("AvgPacketSize", AVG_PACKETSIZE);
        pHeader->SetPropertyULONG32("StartTime",      0);
        pHeader->SetPropertyULONG32("Preroll",        0);
        pHeader->SetPropertyULONG32("Duration",       2000);
        pHeader->SetPropertyBuffer ("StreamName",     NULL);
        pHeader->SetPropertyBuffer ("MimeType",       pMimeType);
        pHeader->SetPropertyULONG32("PopupWindow",       1);
```

Notify the server that the stream header is ready and release memory objects.

```
        m_pFFResponse->StreamHeaderReady(status, pHeader);
        pHeader->Release();
        pMimeType->Release();

    }
}
```

11.7.3 Let the Streaming Begin!

At this stage the File-Format plug-in is ready to generate data packets for delivery over the Internet. The server calls the *GetPacket()* function to read one data packet. The File-Format plug-in requests a read of the data, and when ready, inserts it into an IRMAPacket object along with the time stamp. It also informs the server that the packet is ready through the *IRMAFormat-Response::PacketReady()* function.


```
STDMETHODIMP CExampleFileFormat::GetPacket(UINT16 unStreamNumber)
{
    if (m_state == Ready) {
        m_state = GetPacketReadPending;
        m_pFileObject->Read(PACKET_SIZE);
    }
}
```

PN_RELIABLE_:	specifies the priority level of the packet.
REQUIRED	must be sent or the entire stream will be aborted; for example, renderer initialization information.
HIGH	must arrive or there will be serious problems in the presentation; for example, *Key Frames.*
NORMAL	normal priority; for example, *Audio packets.*
LOW	lower priority; for example, *Images.*
VERY_LOW	can be sent only if there is enough resources available (Server load, network bandwidth, etc).

```
STDMETHODIMP
CExampleFileFormat::ReadDone(PN_STATUS      status, IRMABuffer *pBuffer)
{
```

If the read failed, notify the server that we reached end of stream.

```
    if (status != PN_STATUS_OK) {
        m_ulCurrentTime = 0;
        m_pFFResponse->StreamDone(0);
    }

    if (m_state == GetPacketReadPending) {
        m_state = Ready;
        IRMAPacket* pPacket;
```

Create an IRMAPacket and attach the packet data to it along with the priority level and the time stamp. Then increment the time stamp. *(Continued next page)*

Create an IRMAPacket and attach the packet data to it along with the priority level and the time stamp. Then increment the time stamp.

```
        m_ulCurrentTime += TIME_PER_PACKET;
        m_pFFResponse->PacketReady(status, pPacket);
        pPacket->Release();
    }
}
```

The server calls the *GetPacket()* function repeatedly until the end of file or stream is reached.

11.8 Building a Rendering Plug-in

Rendering plug-ins are responsible for decoding the data, if compressed, and sending it to its final destination—screen, audio device, and so forth. Rendering plug-ins run on the client side and accept data from the Internet in small packets—IRMAPackets.

After the base plug-in interface IRMAPlugin is initialized, the client application calls the *GetRendererInfo()* function to retrieve a list of the supported MIME types. In order to render a stream, the MIME stream type must match one of the supported renderer MIME types (see the File-Format plug-in function *GetStreamHeader()* to learn how to set the MIME type of a stream).

In addition to the MIME types, the *GetRendererInfo()* function returns the rendering refresh rate. This specifies how often you want to render the final data to its destination—screen, audio device, and so forth.

```
PCCHAR CExampleRenderer::zm_pStreamMimeTypes[] = {"application/x-yourRenderFormat",
NULL};

STDMETHODIMP CExampleRenderer::GetRendererInfo(
    REF(const char**)  pStreamMimeTypes,
    REF(UINT32)        unInitialGranularity
    )
{
    pStreamMimeTypes = zm_pStreamMimeTypes;
    unInitialGranularity = 100;
    return S_OK;
}
```

Causes the *OnTimeSync()* function to be called every 100 milliseconds.

The *StartStream()* function is then called to initialize the stream. The function receives a pointer to the stream, IRMAStream, and a pointer to the player, IRMAPlayer. IRMAStream allows you to retrieve information about the stream such as the stream number, type, and input source file. It also allows you to set the quality of the playback, request additional buffered packets, and adjust the refresh granularity. IRMAPlayer gives access to player-related information. It also allows you to start, stop, and seek the stream.

```
STDMETHODIMP
CExampleRenderer::StartStream(
    IRMAStream* pStream,
    IRMAPlayer*    pPlayer
    )
{
    m_pStream  = pStream;
    m_pPlayer  = pPlayer;

    if (m_pStream ) m_pStream->AddRef();
    if (m_pPlayer ) m_pPlayer->AddRef();

    return S_OK;
}
```

IRMAStream functions:
GetSource()
GetStreamNumber()
GetStreamType()
ReportQualityOfService()
ReportRebufferStatus()
SetGranularity()

IRMAPlayer functions:
GetClientEngine()
IsDone()
GetCurrentPlayTime()
OpenURL()
Begin(), Stop(), Pause()
Seek()

The *OnHeader()* function receives the indexed list prepared earlier by the *OnStreamHeader()* function of the File-Format plug-in. You can use the *GetPropertyBuffer()* and *GetPropertyULONG32()* functions to retrieve the information from the indexed list.

```
STDMETHODIMP CExampleRenderer::OnHeader(IRMAValues* pHeader)
{
    // Keep this for later use...
    m_pHeader = pHeader;
    m_pHeader->AddRef();

    // Get the packet data buffer.  Of course you can get more packet info.
    IRMABuffer *pBuffer;
    pHeader->GetPropertyBuffer ("OpaqueData", pBuffer);
    LPBYTE pBuf = pBuffer->GetBuffer();

    return S_OK;
}
```

Once the stream is initialized, the Rendering plug-in receives data packets through the *OnPacket()* function. You can choose to render the data in the packet or wait for the next refresh timer tick—in *OnTimeSynch()* discussed later. In our case, we just save a reference to the packet.

```
STDMETHODIMP CExampleRenderer::OnPacket(IRMAPacket* pPacket)
{
    // Release the last packet if we had one...
    if (m_pLastPacket)
        m_pLastPacket->Release();

    // Keep this one for later use...
    m_pLastPacket = pPacket;

    if (m_pLastPacket)
        m_pLastPacket->AddRef();

    return S_OK;
}
```

When the user starts playing for the first time or resumes playing after a pause, the *OnBegin()* function is called with the stream's time stamp for the next packet.

```
STDMETHODIMP CExampleRenderer::OnBegin(ULONG32 ulTime)
{
    return S_OK;
}
```

To maintain a smooth playback, the RealMedia engine pre-loads extra packets just in case the network gets congested. The RealMedia engine requests additional buffers when the stream starts playing, when the position of the stream is changed (when you seek forward/backwards), or when the number of reserved packets becomes very low. Of course, this action requires CPU cycles both on the server and the client. The client can relinquish CPU cycles by dropping frames or reducing the quality of the final output. This is exactly why the renderer function *OnBuffering()* is called—so that the renderer can adjust the amount of CPU cycles it uses accordingly.

```
STDMETHODIMP
CExampleRenderer::OnBuffering(ULONG32 ulFlags, UINT16 unPercentComplete)
{
    return S_OK;
}
```

When the user pauses playback, the *OnPause()* function is called. This function is typically used to display a static video frame when the movie is paused.

```
STDMETHODIMP CExampleRenderer::OnPause(ULONG32 ulTime)
{
    return S_OK;
}
```

When the user decides to seek forward or backward into the stream, the *OnPreSeek(), OnPostSeek()* functions are called. Both pass the time stamp of the packet before and after the seek.

```
STDMETHODIMP
CExampleRenderer::OnPreSeek(ULONG32 ulOldTime, ULONG32 ulNewTime)
{
    return S_OK;
}

STDMETHODIMP
CExampleRenderer::OnPostSeek(ULONG32 ulOldTime, ULONG32 ulNewTime)
{
    return S_OK;
}
```

The *OnTimeSync()* function is periodically called according to the granularity set in the *GetRendererInfo()* function. Notice that when the player is handling multiple streams, the refresh rate is the same for all streams, and it is equal to the lowest granularity rate of all the streams. The *OnTimeSync()* function is called to update the screen or send data to the audio device.

```
STDMETHODIMP CExampleRenderer::OnTimeSync(ULONG32 ulTime)
{
    // Here's a good time to actually render the data!
    m_ulLastTime = ulTime;

    // Redraw the window.  DamageRect() is similar to the Win32
InvalidateRect()
    CPNxRect    rect(0,0,400,100);
    m_pWindow->DamageRect(rect);

    return S_OK;
}
```

The *UseWindow()* function is called to inform the renderer that it should use a particular window to draw its data, for example, the browser's window. Since RealMedia supports multiple platforms, you should use the platform-independent window interface IRMASimpleWindow. This interface provides platform-independent functions to perform common operations on a window. To get access to the window handler routine, you need to subclass the window and hook into its windows procedure. To simplify cross-platform development, you can use the class PNxSubclassing-Window, which handles platform-independent window subclassing and painting to the window (you can find the definition of this class in RealMedia's sample directory).

In our case, we only set the size of the window and make it visible.

```
STDMETHODIMP CExampleRenderer::UseWindow(IRMASimpleWindow* pWindow)
{
    HRESULT hRes = PNxSubclassingWindow::UseWindow(pWindow);

    // Set the size and visibility of the window
    if (hRes == S_OK) {
        CPNxSize size(400,100);
        hRes = pWindow->SetSize(size);
        pWindow->SetVisibility(TRUE);
    }
    return hRes;
}
```

The window procedure handler of the PNxSubclassingWindow calls the member function *Draw()* to do the painting on the screen. You must override this function to render the output to the window. Since our packets are simple text, we use the *TextOut()* function to display the text in the window.

```
void CExampleRenderer::Draw()
{
    IRMABuffer*pBuffer = NULL;

    if (m_pLastPacket) {
        pBuffer = m_pLastPacket->GetBuffer();
        ::TextOut(m_hDC,0,0,pBuffer->GetBuffer(),pBuffer->GetSize());
    }
}
```

Finally, when the player closes the file, you must release the subclassing window, which in turn destroys the window.

```
STDMETHODIMP CExampleRenderer::ReleaseWindow(IRMASimpleWindow* pWindow)
{
    return PNxSubclassingWindow::ReleaseWindow(pWindow);
}
```

11.9 RealMedia Audio Services

As part of the goal of platform independence, RealMedia defines a hardware-independent interface that provides the necessary methods to deliver audio data to the audio device and to control that device's components—volume, sample bit rate, mono versus stereo, and so forth. This interface is called the *Audio Services* interface.

In addition to hardware independence, the Audio Services interface provides audio mixing capabilities so that multiple audio streams can be mixed together before they're sent out to the audio hardware. It also allows Rendering plug-ins to process the output data of each stream before mixing and to process the final audio data after mixing (see "Touching the Audio Data Before and After Mixing").

Notice that RealMedia comes with a few built-in renderers that can handle RealAudio, WAVE, AU, AIFF, and SND audio file formats. So, you wonder, "Why should I care about Audio Services?" Typically, if you're only dealing with RealAudio or any of the audio formats we have mentioned previously, you don't have to worry about Audio Services. But if you need to handle a new audio format, MPEG audio for example, you will need to use the Audio Services interface to write your data to the audio device.

"Well, why not use DirectSound or RSX?" For one, the Audio Services interface is easy to use. Although DirectSound and RSX provide mixing capabilities, RealMedia's Audio Services interface does its own mixing to maintain platform independence. As a result, if you use RSX or DirectSound, your audio stream will not be mixed with other RealMedia audio streams. Moreover, since the Audio Services interface is platform independent, you can easily provide versions of your custom plug-ins on multiple platforms.[5]

5. DirectSound and RSX are audio technologies from Microsoft and Intel. They are discussed in a later section.

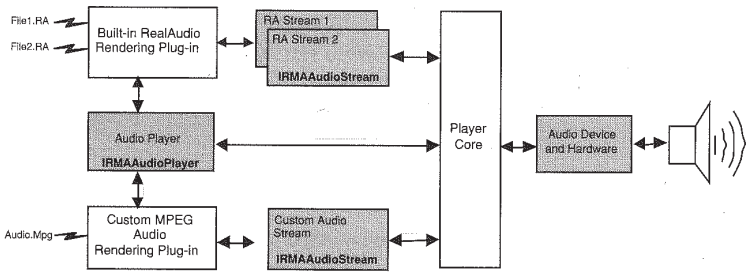


FIGURE 11-4 The RealMedia Audio Services interface in highlighted blocks.

In Figure 11-4, you can see two Rendering plug-ins: the built-in RealAudio Rendering plug-in, which handles RealAudio streams, and a custom MPEG audio Rendering plug-in, which could handle MPEG audio streams. Notice that the Audio Services are shown in the highlighted blocks. The Audio Services consist of one audio player, one audio device, and "multiple" audio streams—one for each active audio stream.

The audio player exposes the IRMAAudioPlayer interface, which allows you to create an audio stream, IRMAAudioStream. The audio stream interface allows you to write the data to the audio device. When you're playing multiple audio streams, the Audio Services mixes them together, including sample rate conversion, and sends the mixed result to the audio device. Finally, the audio device object writes the data to the audio hardware. This is where the platform independence happens.

11.9.1 Playing a Simple Pulse Coded Modulation (PCM) Audio File

Now, let's see how you can use the Audio Services to play an audio file from your custom Rendering plug-in. In this example, we'll only show you how to play a PCM file locally. PCM is the audio format that is typically sent to the audio device. We'll play the audio file whenever the mouse is clicked within the client window. This is a good time to run the demo corresponding to this chapter on the CD.

For the sake of simplicity (or our laziness) let's build on the Rendering plug-in that we've discussed earlier in this chapter. In addition to the text stream, we'll play an audio file, *frog.pcm*, whenever the user clicks the mouse in the client window. You'll know what we're talking about if you've run the demo.

We start by reading the entire PCM file into memory; it is small enough in this case. Since we'll always use the same PCM file, we might as well read it up front in *InitPlugin()*. We allocate an IRMABuffer big enough to hold the entire file and then read the file into it. You learned how to allocate buffers earlier in the chapter.

```
STDMETHODIMP CExampleRenderer::InitPlugin(IUnknown* pContext)
{
    m_pContext = pContext;
    m_pContext->AddRef();
    pContext->QueryInterface(IID_IRMACommonClassFactory, &pClassFactory);

    ULONG32 actual = 0;
    IRMACommonClassFactory* pClassFactory = NULL;

    // allocate an IRMABuffer big enough to hold the entire PCM file
    pClassFactory->CreateInstance(CLSID_IRMABuffer,(void**)&m_pBuffer);
    m_pBuffer->SetSize(LENGTH);

    // Read the file into the buffer and re-adjust its size to the length
    // of the file
    m_pFile = ::fopen(PcmFileName, "rb");
    actual = ::fread(m_pBuffer->GetBuffer(), 1, LENGTH, m_pFile);
    m_pBuffer->SetSize(actual);

    pClassFactory->Release();
    ::fclose(m_pFile);
}
```

You should then create the audio stream when you receive the header information of the stream. If the stream were coming over the Internet, the header would tell you about the stream's properties. Before you can create an audio stream, you must first retrieve a pointer to the audio player interface IRMAAudioPlayer. To do that, you must call the *IRMAPlayer::QueryInterface()* function to get that pointer. Now, you can call the *IRMAAudioPlayer::CreateAudioStream()* function to create the audio stream. If it is successful, you must call the *IRMAAudioStream::Init()* function to initialize the audio stream, specifying mono/stereo, the sampling rate, and the maximum sample size.

```
STDMETHODIMP CExampleRenderer::OnHeader(IRMAValues* pHeader)
{
    // Keep this for later use...
    m_pHeader = pHeader;
    m_pHeader->AddRef();
```

```
    // NOTE: we got a pointer to m_pPlayer in the StartStream() function
    // Get a reference to the IRMAAudioPlayer interface
    m_pPlayer->QueryInterface(IID_IRMAAudioPlayer, &m_pAudioPlayer ))

    // Now create an audio stream and initialize it...
    RMAAudioFormat AudioFmt;

    m_pAudioPlayer->CreateAudioStream(&m_pAudioStream);
    AudioFmt.uChannels = 1;
    AudioFmt.uBitsPerSample = 16;
    AudioFmt.ulSamplesPerSec = 22050;
    AudioFmt.uMaxBlockSize  = (UINT16)LENGTH;
    m_pAudioStream->Init( &AudioFmt, pHeader);
}
```

The *OnMouseClick()* function will be called whenever the mouse is clicked within the client window. It's time to play the file. Since the audio data is in PCM format, we can just write it to the audio device. To do that, we must call the *IRMAAudioStream::Write()* function to send the data to the audio device. Remember: we're actually handing the data over to RealMedia Audio Services, not to the audio hardware. Behind your back, the Audio Services mixes the data with other streams before it sends it out to the audio device object, which sends it out to the audio hardware.

Notice that we set the `ulAudioTime` to a value returned by *GetInstantTime()*. This allows us to play the file instantaneously, so we don't have to wait for it. If you'd rather have the sample be delayed before it is played, just set the time to a relative number in milliseconds. Refer to the RealMedia SDK for more details about instant time and midstream playback.

```
STDMETHODIMP CExampleRenderer::OnMouseClick()
{
    RMAAudioData AudioData;

    // Fill the AudioData structure with a pointer to the data and
    // when it should be played...  In our case, instantaneously.
    AudioData.pData = m_pBuffer;
    AudioData.ulAudioTime = m_pAudioPlayer->GetInstantTime();
    m_pAudioStream->Write(&AudioData);
}
```

11.9.2 Pump Up the Volume

With RealMedia you can also adjust the audio volume at three data points: the output of individual streams, the output of the mixed streams, and the audio device hardware. As you can see in Figure 11-5, each individual

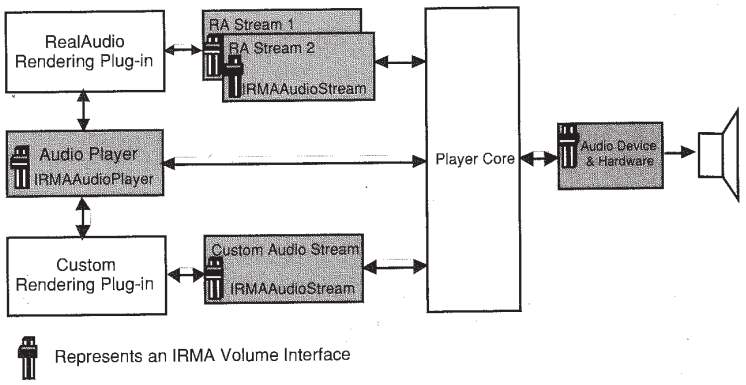


FIGURE 11-5 Volume control for indiviual streams, mixed streams, and audio hardware.

stream has its own volume control. The audio player controls the audio level of the mixed stream, and the audio device object controls the volume of the audio device.

To adjust the volume of an individual stream, you must first call the *IRMAAudioStream::GetStreamVolume()* function, which returns a pointer to a volume interface, IRMAVolume. You can call the *IRMAVolume::SetVolume()* and *GetVolume()* functions to set/get the volume of individual audio streams. A volume setting of 100 means 100 percent of the input signal; values less than 100 reduce the volume, and values greater than 100 increase the volume.

IRMAVolume functions:
Init()
SetVolume(),
GetVolume()
GetLevel()
SetMute()
GetMute()
AddAdviseSink()
RemoveAdviseSink()

```
STDMETHODIMP CExampleRenderer::OnStreamVolume()
{
    IRMAVolume *pVolume = m_pAudioStream ->GetStreamVolume();
    pVolume->SetVolume(90);      // Decrease the volume
    pVoluem->GetVolume(110);     // Increase the volume
    pVolume->Release();
}
```

Similarly, you can adjust the audio level after the streams have been mixed. You must first call the *IRMAAudioPlayer::GetAudioVolume()* function to retrieve a pointer to the IRMAVolume interface. Again, you can call the *IRMAVolume::SetVolume()* and *GetVolume()* functions to set/get the volume of individual audio streams. A volume setting of 100 means 100 percent of the input signal; values less than 100 reduce the volume, and values greater than 100 increase the volume.

```
STDMETHODIMP CExampleRenderer::OnMixedOutputVolume()
{
    IRMAVolume *pVolume = m_pAudioPlayer->GetAudioVolume();
    pVolume->SetVolume(90);     // Decrease the volume
    pVoluem->GetVolume(110);    // Increase the volume
    pVolume->Release();
}
```

The audio device volume is also controlled by an IRMAVolume object, yet here the volume values have a slightly different meaning. In this case, a volume setting of 0 means no sound, and a volume setting of 100 is the maximum volume for the audio hardware.

```
STDMETHODIMP CExampleRenderer::OnAudioDeviceVolume()
{
    IRMAVolume *pVolume = m_pAudioPlayer->GetDeviceVolume();
    pVolume->SetVolume(90);     // Decrease the volume
    pVoluem->GetVolume(110);    // Increase the volume
    pVolume->Release();
}
```


TOUCHING THE AUDIO DATA BEFORE AND AFTER MIXING

RealMedia Audio Services allows you to access the data of individual streams before mixing and after mixing. You can use this mechanism to view or modify the data before sending it off to the audio hardware. For example, you can use the data to show an audio waveform of the data of each individual stream or the data of the mixed stream. You can even choose to modify the data before sending it to the audio hardware. For example, you can use Intel's RSX to add 3D effects to the audio data (see section 13.5 "Mixing Many WAV Files" for more details on 3D audio).

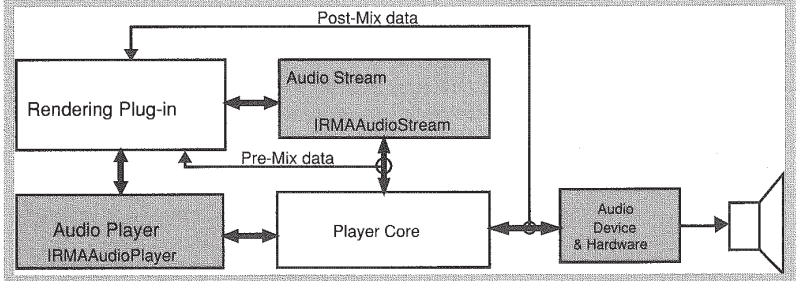


FIGURE 11-6 Using the Rendering plug-in to view or modify audio data before and after mixing.

As you can see in Figure 11-6, a plug-in can interact with the Audio Services to retrieve the audio data of each stream before mixing and after mixing. A plug-in can optionally modify the data before sending it back to Audio Services. For more detail and samples on this topic, refer to the RealMedia SDK.

WHAT HAVE YOU LEARNED?

In this chapter you've learned about the RealMedia technology for real-time streaming of data over the Internet. The data is not limited to audio and video. RealMedia is easily adaptable to stream any custom data type.

At this stage, you should

- be familiar with the concept of RealMedia plug-ins and the different types of plug-ins (File-System, File-Format, and Rendering),
- be familiar with the plug-in interfaces,
- be able to build File-Format and Rendering plug-ins,
- understand the Audio Services interface and how to use it to play additional local audio streams, and
- know how to use Audio Services to adjust the volume of individual streams, mixed streams, and the audio device.

PART IV

Playing and Mixing Sound with DirectSound and RSX 3D

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO JANICE CLEARY, KEVIN O'CONNELL, MICHELLE MCNEIL, RACHEL TILLMAN, CAROL BARRETT, CHRIS ROTVIK, AND TIM ROPER.

Chapter 12 **Audio Mixing with DirectSound**

- Overview of Microsoft's DirectSound
- Play and mix WAV files with DirectSound
- Control final output format and final volume

Chapter 13 **Realistic 3D Sound Experience: RSX 3D**

- Overview of Intel's RSX 3D
- Play and mix audio files with RSX 3D
- Use RSX 3D for real-time 3D sound experience
- Apply reverberation and Doppler effects

Part IV contains quick chapters on two audio architectures for Windows 9x: Microsoft's DirectSound and Intel's 3D Realistic Sound Experience (3D RSX).

Microsoft's DirectSound was designed to address two key performance problems that arose with high-performance multimedia applications running under Windows 95: First, the per-channel overhead for mixing audio channels can be high. Second, there is a noticeable lag between when you request a sound to be played and when it is actually delivered through the speakers. In Chapter 12, we introduce you to Microsoft's DirectSound architecture and show you how to program with it. We show you how to mix and play WAV files and how to control output volume and formats.

Intel's 3D RSX is an architecture that lets listeners perceive sound in all directions, not only to the front and sides, but also above, below, and to the rear. 3D RSX uses just two speakers

(or a set of headphones) to produce a surround sound experience. The system creates sounds based on simulations of how the human brain hears sounds and is based on a Head Related Transfer Function (HRTF) technology. In Chapter 13, we will give you an overview of 3D RSX, show you how to play audio files with it, and how to add special effects to your sounds, such as reverberation or Doppler effects.

tem creates
based on a
give you an
ld special

CHAPTER 12

Audio Mixing with DirectSound

WHY READ THIS CHAPTER?

Microsoft's DirectSound was designed to address two key performance problems with standard audio for high-performance multimedia applications running on Windows 95: First, the CPU usage for mixing audio channels is high. Second, there is a noticeable lag between when you request a sound to be played and when it is actually delivered through the speakers.

If you've been facing either of these performance problems with your multimedia application and would like to understand DirectSound's solutions, or if you expect the audio component of your application to be demanding and instinctively know that you will need a high-performance audio solution, read on.

By the time you have worked through this chapter, you will

- have a good idea of how DirectSound works and what it offers,
- understand how DirectSound reduces audio latency under Windows 95,
- have learned how to play a WAV file and mix WAV files using DirectSound, and
- have learned how to control output volume and formats through DirectSound.

PART IV

12.1 Overview of Audio under Windows 95

The standard Microsoft multimedia library (previously called *mmsystem.lib* and now called *winmm.lib*) provides developers with a wide range of functions for interacting with audio devices and performing audio functions. These functions range from high-level interfaces for basic audio tasks to low-level interfaces that provide more control of task and audio devices.

The *PlaySound()* function, for example, is a simple high-level way to play audio files or to play audio sounds from the system registry. The MCIWnd class, on the other hand, supplied as part of Video for Windows (*VFW.h*), provides multimedia extensions for Windows. The audio services in MCIWnd provide input, output, and recording control of a variety of devices including CD audio, WAV audio, MIDI, and audio-video devices.

Table 12-1 lists the categories of audio services provided in the standard Windows multimedia library.

TABLE 12-1 Range of Audio Services Available in Standard Windows Multimedia Library

Prefix	Service
wave	Works with sounds in the PCM waveform audio format. In addition to playing audio sounds, *wave* functions provide for audio input, for audio record, and for waveform audio device control.
midi	Plays and records Musical Instrument Digital Interface (MIDI) sound representations. The MidiMapper (with channel maps, patch maps, and key maps) provides a device-independent interface for playing MIDI files.
mixer	Provides runtime mixing of multiple MIDI or multiple WAV audio streams within a single application.
mci	Media Control Interface controls a variety of multimedia devices including audio devices such as CD Audio, WAV audio, MIDI, and audio-video.
acm	The Audio Compression Manager is an extension of the basic multimedia system that enables runtime audio compression, decompression, and filter services.

12.2 DirectSound Features

There are two performance problems with the audio services in *winmm.lib:*

- The performance overhead of mixing audio streams is high. On baseline 90 MHz Pentium platforms, mixing eight audio sounds consumes at least 40 percent of the CPU, leaving very little capacity for even more performance intensive tasks such as running graphics. The overhead increases when audio formats differ and format conversion is necessary.
- The latency between when an application plays a sound and when the sound is delivered through the speakers can be between 100 to 150 milliseconds. Consider comparing an audio sound to a graphics event like crashing into a wall. With a latency of 100 milliseconds and a frame rate of 30 frames per second, at least 3 frames have gone by before the sound is heard. The graphics actually seen would probably have no bearing on

the sound. For adequate synchronization, the audio sound must be heard before the next frame is drawn (less than 33 milliseconds at 30 fps).

In Microsoft's words: "The overriding design goal in DirectX is speed." The Microsoft DirectSound audio library provides high-performance, low-overhead, low-latency audio mixing. DirectSound accesses hardware acceleration whenever possible. In addition, the DirectSound architecture gives Windows applications direct access to the sound device.

In addition to solving performance problems, the DirectSound component of the DirectX SDK adds a notable feature: It can mix audio streams from multiple applications. You can design your DirectSound-based application to allow mixing sounds from other DirectSound applications. With this feature, for example, a DirectSound-based Internet-audio-phone can share audio output with your DirectSound application.

12.3 DirectSound Architecture

Figure 12-1 shows how DirectSound fits into the Windows 95 audio framework. As part of the DirectSound architecture, Microsoft defined extensions to the standard Windows 95 audio device driver. The extended interface is known as the DirectSound Hardware Abstraction Layer (HAL) interface. DirectSound provides its enhanced performance and features via

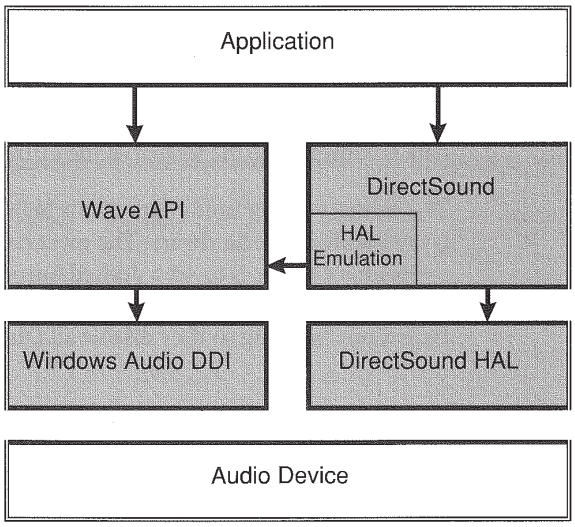


FIGURE 12-1 DirectSound architecture within the Windows 95 framework.

the DirectSound HAL. The DirectSound path and the WAV audio path are two mutually exclusive paths to the audio hardware. They cannot be used simultaneously.

If a DirectSound driver is not available, DirectSound will use the standard Windows 95 audio device driver. In this case, DirectSound provides neither low-latency audio nor device access; but it can still provide low-overhead mixing.

To us application developers, DirectSound provides `DIRECTSOUND` objects as representatives of audio cards in the system. We access all further DirectSound functionality through these `DIRECTSOUND` objects. Our applications can only instantiate one DirectSound object per device. But multiple applications can each instantiate their own DirectSound objects, and the application in focus will have principal control of the audio output. (*DirectSoundCreate()* and *DirectSoundEnumerate()* are the only two functions that can be called without having instantiated a `DIRECTSOUND` object.)

To create and play sounds, DirectSound provides two types of `DIRECTSOUNDBUFFER` objects: secondary and primary. Secondary `DIRECTSOUNDBUFFER` objects represent individual sounds or sound streams. DirectSound mixes individual secondary buffers into the primary `DIRECTSOUNDBUFFER`, and this mixed data is sent to the audio device.

Secondary buffers can be either hardware or software. Hardware buffers are created and used if the audio device supports hardware mixing. Hardware mixing reduces system overhead cost. In the absence of hardware mixing, system buffers and software mixing are used with some CPU overhead required for the mixing. Data in secondary buffers can be of varying audio formats. All data is converted to the format of the primary buffer during mixing.

The primary buffer holds data that is being played by the audio device and is invariably in the hardware. The most common model used for accessing the primary buffer is to set a desired output format or to control total output volume. Additionally, applications can write directly to the primary buffer, but by doing so they disable all DirectSound mixing.[1]

---

1. We suggest that you avoid writing directly to the primary buffers. The size of audio buffers is dictated by DirectSound device drivers. Buffers must be filled on time, as gaps in audio buffers are heard as annoying audio clicks. Timer and thread management becomes fairly complex with small buffers, and on the other hand, latency is high with large buffers.

Through an esoteric but not atypical way of using primary buffers you can reduce latency when you are playing individual short sounds. It works as follows. Null data in the primary buffer is played in LOOP_MODE, which forces data to be sent to the audio device constantly. When sound in a secondary buffer is played, data is merely mixed into the already playing buffer, and there is no initialization latency delay.

12.4 Playing a WAV File Using DirectSound

12.4.1 Initializing DirectSound

First, we need to initialize DirectSound. The starting point for using DirectSound is the *DirectSoundCreate()* function call. We get access to all DirectSound functions through the DIRECTSOUND object that *DirectSoundCreate()* instantiates.

Right after initializing the DIRECTSOUND object, we've got to establish how we plan on using DirectSound by using *IDirectSound::SetCooperativeLevel()*. DirectSound has four cooperative levels: DSSCL_NORMAL, DSSCL_PRIORITY, DSSCL_EXCLUSIVE, and DSSCL_WRITEPRIMARY.

The DSSCL_NORMAL cooperative level is sufficient for our current example of simply playing a WAV file with DirectSound. DSSCL_NORMAL sets up our use of DirectSound for smooth audio sharing with other applications; note that the final output format is automatically fixed to 8-bit, 22-kHz STEREO format, and no format conversions are required when the focus switches.

Here's the code to initialize access to the audio device through DirectSound:

```
BOOL CSharedHardware::Init(HWND hWnd) {
    LPDIRECTSOUND   pDSound;
    HRESULT         err;

    // create a DirectSound instance
    DirectSoundCreate(NULL, &pDSound, NULL);
```

DirectSoundCreate() is the starting point in using DirectSound. The DIRECTSOUND structure returned from this function provides access to the next level of functionality, such as *CreateSoundBuffer*, *GetCaps*, and so forth.

```
    // Setup to use as normal windowed app
    err = pDSound->SetCooperativeLevel(hWnd, DSSCL_NORMAL);
    if (err != DD_OK) {
        pDSound->Release();
        return FALSE;
    }

    // store away
    m_pDSound = pDSound;
    // return success code
    return TRUE;
}
```

SetCooperativeLevel() sets how we plan to use DirectSound. DirectSound permits four different levels of usage:

DSSCL_NORMAL	Most cooperative, smoothest resource sharing with other applications. However, output format is fixed to 8-bit, 22-KHz, mono.
DSSCL_PRIORITY	At this priority level, the application can change the output format.
DDSCL_EXCLUSIVE	At this level, sounds from other apps are not heard when this app has the input focus. Output format can be set.
DDSCL_WRITEPRIMARY	Application gets direct access to the primary output buffers. However, secondary buffers cannot be played and application must do its own mixing.

12.4.2 DirectSound Structures

To play sounds using DirectSound, we need to set up the sound in a DirectSound format. DirectSound's DSBUFFERDESC structure defines the format for sound buffers. The actual format of the sound data is defined using the standard WAVEFORMATEX structure from *mmreg.h*. Let's take a look at these structures.

```
typedef struct _dsbufferdesc{
    DWORD           dwSize;
    DWORD           dwFlags;
    DWORD           dwBufferBytes;
    DWORD           dwReserved;
    LPWAVEFORMATEX  lpwfxFormat;
} DSBUFFERDESC, *LPDSBUFFERDESC;

typedef struct {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD nSamplesPerSec;
    DWORD nAvgBytesPerSec;
    WORD  nBlockAlign;
    WORD  wBitsPerSample;
    WORD  cbSize;
} WAVEFORMATEX;
```

- dwBufferBytes should indicate the size of the sound buffer. The application sets this field for secondary buffers, and DirectSound sets this field for primary buffers.
- Actual sound format is defined using the standard WAVEFORMATEX structure defined in mmreg.h. We've included the definition here for quick reference.

The dwFlags field is used both to establish type of sound buffer being created as well as to describe attributes upon return. Refer to the DirectSound documentation for more details. Some interesting flags are

_DSBCAPS_PRIMARYBUFFER	Request a primary buffer. If this flag is not set, a Secondary buffer will be created.
_DSBCAPS_STATIC	Sound will be used repeatedly. Designate as good candidate for hardware acceleration.
_DSBCAPS_LOCHARDWARE	Forces the buffer to be in hardware memory.
_DSBCAPS_LOCSOFTWARE	Forces the buffer to be in system memory.

Some additional flags to control special effects that we leave for extra credit exploration are _DSBCAPS_CTRLALL, _DSBCAPS_CTRLDEFAULT, _DSBCAPS_CTRLFREQUENCY, _DSBCAPS_CTRLPAN and _DSBCAPS_CTRLVOLUME. We will use _DSBCAPS_CTRLVOLUME later in this chapter.

12.4.3 Creating Sound Buffers

We've successfully initialized DirectSound. Let's create a simple secondary DIRECTSOUNDBUFFER.

```
BOOL CSound::Init(LPDIRECTSOUND pDSound, LPSTR lpszFileName)
{
    HRESULT err
    // first try load wave file into memory
    if (!LoadWavFile(lpszFileName)) return FALSE;

    // Create a device sound buffer
    m_dsDesc.dwFlags = DSBCAPS_STICKYFOCUS;
    m_dsDesc.dwBufferBytes = m_dwSizeData;
    m_dsDesc.lpwfxFormat = m_pWavFmt;
    pDSound->CreateSoundBuffer(&m_dsDesc, &m_pBufferFns, NULL);

    ///////// transfer data from memory to dsBuffer.
    // first lock the entire buffer.
    LPVOID pBlk1, pBlk2;              // dsound maintains split-buffers
    DWORD dwSize1, dwSize2;           // size of each buffer
    m_pBufferFns->Lock(0,m_dwSizeData,&pBlk1,&dwSize1,&pBlk2,&dwSize2,0);
    // write data into possibly 2 buffers that DirectSound returns
    CopyMemory(pBlk1, m_pSrcData, dwSize1);
    if (dwSize2 > 0)
        CopyMemory(pBlk2, m_pSrcData+dwSize1, dwSize2);
    // unlock both buffers and return
    m_pBufferFns->Unlock(pBlk1, dwSize1, pBlk2, dwSize2);
    return TRUE;
}
```

We wrote a simple WavFile Load routine based on standard *mmio* calls in the *winmm* multimedia library. Upon return, this function will load WAVEFORMATEX, bufferSize, and bufferData into member variables.

- *CreateSoundBuffer()* takes LPDSBUFFERDESC and (LPDIRECTSOUNDBUFFER *) as parameters. We describe the surface that we're requesting in LPDSBUFFERDESC. If the Create is successful LPDIRECTSOUNDBUFFER points to the member functions of the created Sound Buffer.
- The only flag we specified was DSBCAPS_STICKYFOCUS, which will let our sounds be played even if we're not the application with the input focus. However, if we lose focus to another DirectSound application, we will lose our audio output.
- DSBCAPS_GLOBALFOCUS in DirectX allows our sounds to continue playing even if we lose focus to another DirectSound application. However, cooperative levels of DDSCL_EXCLUSIVE will override even the DSBCAPS_GLOBALFOCUS setting.
- WaveFormat and BufferLength are set to the values returned by the *LoadWavFile()* routine.

DirectSound sees sound buffers with a circular reference pattern. Circular views enables "infinite" streaming buffers: as the front of the buffer is being consumed, the rear of the buffer can be refilled. Circular views also make it easy to implement looped sounds for "static" fixed size buffers. *Locks()*, *Unlocks()*, and data access with circular views use two buffer access descriptors, where a buffer access descriptor is a (pointer, size) combination.

12.4.4 Playing the Sound

Now that we've created our sound buffer, playing the sound is as simple as invoking the *IDirectSoundBuffer::Play()* member function.

```
BOOL CSound::Play()
{
    m_pBufferFns->Play(0, 0, DSBPLAY_LOOPING);
    return TRUE;
}
```

DirectSound requires the first two parameters to *IDirectSoundBuffer::Play()* to be 0. The third parameter allows for flags to control the Play mode. Currently the only flag defined is DSBPLAY_LOOPING; therefore, DirectSound permits sounds to be either PlayedOnce or PlayedForever. Playing a secondary buffer will mix the data from the sound buffer into the primary buffer. *IDirectSoundBuffer::Stop()* can be used to stop sound buffers that are playing.

12.4.5 Demo Time

Run the demo that corresponds to this chapter. Since we have set the sound to be played in LOOP_MODE, you should hear the sound play continuously. Try switching to another application such as the Calculator. You should still hear the sound even though our application has lost the input focus. This is the result of creating the buffer with the DSBCAPS_STICKYFOCUS flag. Try invoking a second instance of our sound application. You will hear only one sound being played.

EXTRA CREDIT

Recompile the application after having set the secondary buffers to be created with the DSBCAPS_GLOBALFOCUS flag. Invoke multiple instances of the application. You should hear multiple sound streams being mixed together.

12.4.6 Mixing Two WAV Files

Mixing two WAV files is as simple as creating another secondary buffer and playing it. DirectSound will automatically mix playing sounds together.

It seems ridiculous to show this code, but we'll do it anyway. Run the demo that corresponds to this chapter and create a second sound to invoke audio mixing.

```
BOOL OnNewSound(LPSTR lpszFileName)
{
    // create new sound
    CSound *pNewSound = new CSound;
    if (!pNewSound->Init(lpszFileName)) return FALSE;

    // start the sound playing
    pNewSound->Play(DSBPLAY_LOOPING);

    // and store handle in list
    gSounds[gnSounds] = pNewSound;
    gnSounds++;
}
```

12.5 Controlling the Primary Sound Buffer

So far we've played a WAV file with DirectSound using the system's default audio format. What if we want to change this output format to work with sound samples of higher (or even lower) quality? To change the output format, or to change the total output volume, we would need to control the primary sound buffer.

12.5.1 Initializing to Get Control of the Output Format

While initializing DirectSound, we need to set the CooperativeLevel to allow us to change output format privileges:

```
BOOL CSharedHardware::Init(HWND hWnd)
{
    LPDIRECTSOUND    pDSound;
    HRESULT          err;
    // create a DirectSound instance
    DirectSoundCreate(NULL, &pDSound, NULL);

    // Setup to use as priority app
    err = pDSound->SetCooperativeLevel(hWnd, DSSCL_PRIORITY);
    if (err != DD_OK) {
        pDSound->Release();
        return FALSE;
    }
    m_pDSound = pDSound;
    return TRUE;
}
```

SetCooperativeLevel() to DSSCL_PRIORITY. At this priority level, the application can change the output format.

12.5.2 Creating a Primary DirectSound Buffer

Now let's create a primary DirectSoundBuffer object so that we can get access to the *IDirectSoundBuffer::SetFormat()* function.

```
BOOL CSoundPrimary::Init(LPDIRECTSOUND pDSound)
{
    // Create a primary sound buffer
    m_dsDesc.dwFlags = DSBCAPS_PRIMARYBUFFER | DSBCAPS_CTRLVOLUME;
    m_dsDesc.dwBufferBytes = 0;
    m_dsDesc.lpwfxFormat = NULL;
    pDSound->CreateSoundBuffer(&m_dsDesc, &m_pBufferFns, NULL);

    // find out the format of the primary buffer
    DWORD dwSizeToAlloc;
    m_pBufferFns->GetFormat(NULL, 0, &dwSizeToAlloc);
    m_pFmt = (WAVEFORMATEX *)(new BYTE[dwSizeToAlloc]);
    m_pBufferFns->GetFormat(&m_pFmt, dwSizeToAlloc, NULL);

}
```

- Set flags to DSBCAPS_PRIMARYBUFFER to request access to the primary buffer. Also set DSBCAPS_CTRLVOLUME flag to allow volume control to be queried and set.
- We cannot specify the size of the primary sound buffer and must set the size to 0.
- Similarly, we cannot specify output format during creation, but can change it using the *SetFormat()* function call; therefore set the waveformat pointer to NULL.

IDirectSoundBuffer::GetFormat() must be called twice. Once with a NULL pointer to find out the size of the buffer to allocate and then with a valid pointer to the buffer just allocated.

Now that we have created a primary DirectSoundBuffer object, we have control over the output format and the total output volume. (Note that the DSBCAPS_CTRLVOLUME must be set to allow the volume control to be modified.) Here is sample code that changes the volume and the output format of the primary buffer.

```
// change the format of the primary buffer
m_pFmt->nSamplesPerSec *= 2;
m_pFmt->nAvgBytesPerSec = m_pFmt->nSamplesPerSec * m_pFmt->nBlockAlign;
pBufferFns->SetFormat(m_pFmt);

// get & set total audio volume
long lVolume;
err = m_pBufferFns->GetVolume(&lVolume);
lVolume = 2 * lVolume;
err = m_pBufferFns->SetVolume(lVolume);
```

For a valid WAVE-FORMAT specification, AverageBytesPerSec must be a product of the SamplesPerSec and the BlockAlignment.

DirectSound does not currently support making sounds louder. The volume returned will be the current attenuation level of total volume. Doubling this already negative value will cause the sound volume to be greatly reduced. If the DSBCAPS_CTRLVOLUME flag was not set, both these calls would have returned a DSERR_CONTROLUNAVAIL error indicating that volume control was not set up during buffer creation.

12.5.3 Demo Time

Run the demo that corresponds to this chapter. Check the primary DirectSoundBuffer option to enable the format and volume controls. Play around with the volume and sample rate controls. In particular, try reducing the sample rate of the output format and see if you can detect a quality degradation. Switch to another application with DirectSound audio *RSXDemoApp* and see if the format/volume changes persist.

WHAT HAVE YOU LEARNED?

By this time, you've had an overview of DirectSound and what it does for you. If you worked through the code samples, you have

- played a WAV file using DirectSound,
- mixed two WAV files using DirectSound, and
- controlled the final output format and volume while your application was in focus.

In the next audio chapter you will be introduced to 3D sounds and special effects using Intel's RSX 3D.

CHAPTER 13

Realistic 3D Sound Experience: RSX 3D

WHY READ THIS CHAPTER?

Now that you've looked at DirectSound, you might be wondering if there is a simple way of playing just a generic sound file without taxing application performance. Intel's Realistic 3D Sound Experience (RSX 3D) library helps you do just that. It also gives you a 3D sound model that mimics the real world environment.

To get the most out of this chapter, we recommend that you run the audio demos on the companion CD while you are reading this chapter or beforehand.

In this chapter you will

- get an overview of RSX 3D features,
- see how simple it is to play and mix two or more audio files,
- learn how to use the RSX 3D sound model to achieve a realistic sound experience, and
- learn how to add reverberation and Doppler effects to your application.

PART IV

Microsoft's DirectSound provides direct access to audio devices under Windows and allows developers to implement low latency audio applications. Even though DirectSound provides some level of abstraction from the hardware, developers must still handle the intricacies of various devices.

Intel's Realistic 3D Sound Experience (RSX 3D) library provides a simple high-level interface for rendering audio under Windows. It implements an abstraction layer above the DirectSound and WAV APIs without sacrificing

audio performance. In addition, RSX 3D introduces a new 3D environment that models the sound's physical properties, making for an immersive experience.

In this chapter, we'll first show you how easy it is to play one or more WAV files using RSX 3D. We'll then give you an overview of the RSX 3D environment and show you how to provide realistic 3D sound in your application. Finally, we'll glance over RSX 3D support for streaming audio data.

13.2

13.1 RSX 3D Features

RSX 3D is a high-performance audio library that provides developers with a simple interface for rendering audio without taxing application performance. Depending on the configuration of your system, RSX 3D uses either the DirectSound or WAV API to access the audio device. Depending on the power of you processor, RSX 3D automatically scales the output to sound better on high-end processors.

RSX 3D's simple interface allows developers to play audio files from either local drives, networked drives, or even across the Internet. The files themselves can be of different formats (WAV or MIDI) and different sample rates.

One of RSX 3D's most exciting features is its new 3D audio environment, which models real 3D graphics environments. RSX 3D can position sounds anywhere in 3D space. The sound may be above your head, below your feet, behind you, in front of you, and so forth.

Say, for example, you are standing in a hallway and a door slams shut to your right. The sound will reach your right ear earlier than it will reach your left ear (this phenomenon is called Interaural Time Delay, ITD). The sound will also be louder in your right ear than in your left ear (this is known as Interaural Intensity Difference, IID). With these cues your brain is able to correctly locate the sound as originating from your right and not from your left. RSX 3D uses these and other cues to produce realistic sounds as objects move around a scene.[1] RSX 3D also supports modifying sounds for special effects including Doppler, reverberation, and pitch calculations. In

1. ITDs and IIDs are combined with other cues to form Head Related Transfer Functions (HRTFs). Clinical probe microphones are inserted into the ears of volunteers to record HRTF measurements. RSX 3D uses HRTF technology and HRTF measurements to simulate 3D sound on PCs.

this chapter, we will work through examples of using RSX 3D's sound positioning capabilities and its special effects capabilities.

13.2 Creating an RSX 3D Object

The RSX 3D audio library uses Microsoft's Component Object Model (COM) interface to export its features. In order to use any COM module and the COM functions you must initialize the COM libraries at start-up time. You can initialize the COM libraries by calling the *CoInitialize()* function. In addition, you need to release any COM objects used by your application and then call *CoUninitialize()* when your application terminates.

To use RSX 3D, you must first create an RSX 3D object within your application process space. You can use *CoCreateInstance()* to create this object, specifying `CLSID_RSX20` in the Class ID field. RSX 3D only supports this in-process creation model where RSX objects are created within the application memory context.

```
// Initialize the COM libraries..
m_coResult = CoInitialize(NULL);
if (FAILED(m_coResult)) {
    AfxMessageBox("Failed to load COM libraries");
    return -1;
}

// Create the RSX20 object and get an IUnknown pointer to the object
HRESULT hr = CoCreateInstance(
    CLSID_RSX20,                ◊ GUID for RSX20. Defined in rsx.h.
    NULL,                       ◊ Create object within processes (only supported mode).
    CLSCTX_INPROC_SERVER,       ◊ Only need to create object and don't care about its methods.
    IID_IUnknown,               ◊ Holds the IUnknown instance of the object.
    (void ** ) &m_lpUnk);

// Make sure that everything is fine..
// If the application fails here, just get out..
//
if( (FAILED(hr)) || (!m_lpUnk) ) {
    AfxMessageBox("Failed to Create RSX Object.\n"
        "CoCreateInstance Failed - Please run RSX Setup\n");
    PostMessage(WM_CLOSE);
    return -1;
}
```

When using COM, make sure to
- define INITGUID before include files,
- link in *OLE32.LIB*, and
- turn off automatic use of precompiled headers.

13.3 Play one WAV file

Once you have created the RSX 3D object, it is a very simple process to play any WAV or MIDI audio file. But before we go into the details, let's first introduce the environment that RSX 3D uses to describe its objects.

To play an audio file with RSX 3D, you need to create only two objects: an *emitter* and a *listener*. I tend to think of an emitter as a jukebox, and a listener as my own ears—just like in the real world. In RSX 3D a *cached emitter* is an object that handles reading and decompressing an audio file, and a *direct listener* is an object that handles sample rate conversion and mixing and writing the output data to the audio device.

So let's create the direct listener first. As with any good COM object, you must use the *CoCreateInstance()* function to create the listener object and pass the `CLSID_RSXDIRECTLISTENER` for the Class ID parameter. Once the object is created, you call the *Initialize()* function to initialize the listener. In all of the initialize calls, notice that you must also pass in an IUnknown pointer to the main RSX object. Also notice that you can only have one listener active within an application and that you can call the initialize function only once throughout the life of a listener.

```
HRESULT CRsxSampleView::CreateDirectListener()
{

    // First, we need to create an instance of the listener object
    HRESULT hr = CoCreateInstance(
        CLSID_RSXDIRECTLISTENER,      ◊ GUID for Direct Listener object. Defined in rsx.h
        NULL,
        CLSCTX_INPROC_SERVER,         ◊ Create object within process context.
        IID_IRSXDirectListener,       ◊ Direct Listener Interface identifier
        (void ** ) &m_lpDL);          ◊ Holds the listener instance


    // If all is fine, you must initialize
    // the Direct Listener interface
    // before you do anything else.
    if(SUCCEEDED(hr) && m_lpDL) {
        RSXDIRECTLISTENERDESC rsxDL ;
        ZeroMemory(&rsxDL, sizeof(rsxDL));

        rsxDL.cbSize = sizeof(rsxDL);
        rsxDL.hMainWnd = m_hWnd;
        rsxDL.dwUser = 0;
        rsxDL.lpwf = NULL;
```

hMainWnd: DirectSound requires a window handle. You can set the Registry Key Device Type to `DIRECTSOUND`.

lpwf: Points to `WAVEFORMATEX` structure which specifies the format of output data. If NULL, RSX uses the default format in the RSX configuration, or in the Registry.

Follow the Registry Settings link in the RSX online help for more details.

```
        hr = m_lpDL->Initialize(&rsxDL, m_lpUnk);
    }
    return hr;
}
```

Similarly, you call the *CoCreateInstance()* function to create the cached emitter with `CLSID_RSXCACHEDEMITTER` in the Class ID field. Before you can call any other method within the emitter, you need to initialize the object by calling the *Initialize()* function.

Notice that, for a cached emitter, you can specify an audio file that exists on a local drive, network drive, or even on a URL, a Web site, or an FTP site. (Note: To use URL-based emitters, Microsoft's Internet Explorer 3.0 or later must be installed and configured on your computer.)

```
HRESULT CRsxSampleView::CreateCachedEmitter(
    LPCTSTR pszFile,
    IRSXCachedEmitter** lppCE)
{
    HRESULT hr = CoCreateInstance(
                CLSID_RSXCACHEDEMITTER,
                NULL,
                CLSCTX_INPROC_SERVER,
                IID_IRSXCachedEmitter,
                (void ** )lppCE);

    if(SUCCEEDED(hr) && *lppCE) {
        RSXCACHEDEMITTERDESC rsxCE;
        ZeroMemory(&rsxCE, sizeof(rsxCE));

        rsxCE.cbSize = sizeof(rsxCE);
        rsxCE.dwFlags = RSXEMITTERDESC_NODOPPLER |
                        RSXEMITTERDESC_NOREVERB |
                        RSXEMITTERDESC_NOATTENUATE |
                        RSXEMITTERDESC_NOSPATIIALIZE;
        rsxCE.dwUser = 0;
        strcpy(rsxCE.szFilename, pszFile);
        hr = (*lppCE)->Initialize(&rsxCE, m_lpUnk);
    }
    return hr;
}
```

dwFlags: You can combine any of the following:

_NODOPPLER:	Disable Doppler effect
_NOATTENUATE:	Disable distance attenuation
_NOSPATIALIZE:	Disable spatial calculations
_NOREVERB:	Disable sound reverberation

szFileName: Path to the audio file:

c:\media\audio.wav	On local or network drive
SomeDll.Dll 108	Resource 108 inside DLL
http://www.xyz.com/media.wav	on Web site

13.4 Play One WAV File

Once the emitter is initialized, you can call the *ControlMedia()* function with `RSX_PLAY` in order to play the file. *All set? So play it, maestro!* Put on your headphones, or crank up your speakers, and enjoy.

```
void CRsxSampleView::OnPlayOneFile()
{
    CreateDirectListener();
    CreateCachedEmitter("file1.wav", &m_lpCE);
    m_lpCE->ControlMedia(RSX_PLAY, 0, 0.0f);
}
```

fInitialStartTime: starting position in seconds.

nLoops: Number of loops to play. 0: infinite

13.5 Mixing Many WAV Files

That was simple, wasn't it? Now let's see what it takes to mix two different audio files together. In real life you only need to put another jukebox in the same room, and you would hear both of them together. Well, the process is very similar in RSX 3D. If you add another emitter to the set, then you'll have two audio files playing at the same time. RSX 3D takes care of mixing them for you and delivering the mixed output to the listener object. That's all it takes!

```
void CRsxSampleView::OnPlayMixtwoaudiofiles()
{
    CreateDirectListener();                          ◊ Only one listener.
    CreateCachedEmitter("file1.wav", &m_lpCE);       ◊ Sound source 1.
    CreateCachedEmitter("file2.wav", &m_lpCE2);      ◊ Sound source 2.
    m_lpCE->ControlMedia(RSX_PLAY, 0, 0.0f);         ◊ Play source 1.
    m_lpCE2->ControlMedia(RSX_PLAY, 0, 0.0f);        ◊ Play source 2.
}
```

13.6 RSX Goes 3D—True 3D Sound

Well, let's pause and think about the model that RSX 3D uses to represent its objects. Recall that RSX 3D mimics the real-world environment, using a sound emitter and a listener to represent its objects. Why not take this a bit further and add positional attributes to these objects (emitter and listener)? That's exactly what RSX 3D does (see Figure 13-1).

Similar to 3D graphics objects, RSX 3D objects can possess positional 3D attributes based on *x,y,z* coordinates. RSX 3D uses the relative 3D position between the listener and emitter(s) to calculate the audio volume for the left and right speaker channels. For example, if you position the emitter

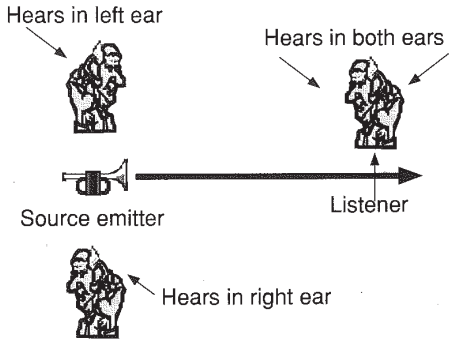


FIGURE 13-1 Physical sound properties.

exactly to the left of the listener, you would hear the sound predominantly from the left speaker channel and in your left ear—just like in real life.

Before we delve into the tiny details, let's look at the model that RSX 3D uses for the emitter. As you know, sound travels the farthest in the direction in which it is pointing, less to the sides, and even less in the opposite direction. You also know that sound volume decreases as you move away from the sound source.

The RSX 3D sound emitter mimics real-world conditions. As you can see in Figure 13-2, RSX 3D defines two ellipses for the emitter, one inside the other. The inner ellipse represents the ambient region where the sound retains maximum intensity and contains no directional information. The outer ellipse defines the region where the sound intensity decreases logarithmically as you move away from the emitter. The emitter does not contribute any of its audio outside the outer ellipse.

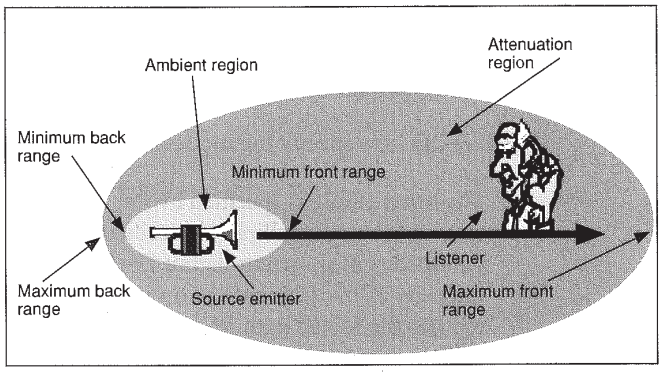


FIGURE 13-2 RSX elliptical sound model.

You define the ellipses by the distance of the front and back ranges from the emitter. Different emitters on the scene could have different sound characteristics; for example, louder emitters would have larger ellipses.

As the relative position of the emitter and the listener changes, the sound characteristics change to mimic the real-world situation. The picture in Figure 3-2 shows 2D ellipses; in RSX 3D the ellipses are actually 3D ellipsoids.

13.7 Setting Up 3D Sound with RSX 3D

Now let's see how we can use RSX 3D to define the 3D position of the objects, emitter and listener, and allow it to deliver a *realistic 3D sound experience.* If you go back to where we created the emitter, you'll notice that we disabled the special effects such as sound attenuation, Doppler effect, and so forth. At that point, we only wanted to play some generic audio file. Now we need these cool effects, so let's go back in and enable them. The boldface line in the following block of code enables all 3D sound effects.

```
HRESULT CRsxSampleView::CreateCachedEmitter(
    LPCTSTR pszFile,
    IRSXCachedEmitter** lppCE)
{
    HRESULT hr = CoCreateInstance( CLSID_RSXCACHEDEMITTER, NULL, CLSCTX_INPROC_SERVER, IID_IRSXCachedEmitter,lppCE);
    if(SUCCEEDED(hr) && *lppCE) {
        RSXCACHEDEMITTERDESC rsxCE;
        ZeroMemory(&rsxCE, sizeof(rsxCE));
        rsxCE.cbSize = sizeof(rsxCE);

        rsxCE.dwFlags =    0;

        rsxCE.dwUser = 0;
        strcpy(rsxCE.szFilename, pszFile);
        hr = (*lppCE)->Initialize(&rsxCE, m_lpUnk);
    }

    return hr;
}
```

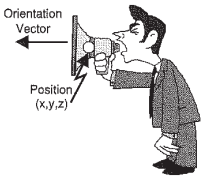
We can now specify the model that describes the behavior of the emitter.

The inner ellipse is specified by the `fMinFront` and `fMinBack` parameters, and the outer ellipse is specified by the `fMaxFront` and `fMaxBack` parameters as shown in Figure 13-2. Finally, you specify the maximum intensity of the ambient region and then call the *SetModel()* function to register the model with RSX 3D.

```
void CRsxSampleView::SetEmitterPosition(IRSXCachedEmitter* lpCE, int x, int y, int z)
{
    // Now we should set the emitter model
    RSXVECTOR3D v3d;
    RSXEMITTERMODEL rsxEModel;
    rsxEModel.cbSize = sizeof(RSXEMITTERMODEL);
    rsxEModel.fMinFront = 100.0f;
    rsxEModel.fMinBack = 100.0f;
    rsxEModel.fMaxFront = 800.0f;
    rsxEModel.fMaxBack = 200.0f;
    rsxEModel.fIntensity = 1.0f;
    lpCE->SetModel(&rsxEModel);
```

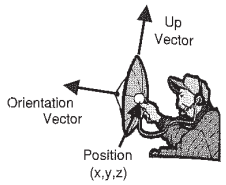
Finally, we need to position the emitter in the 3D scene and specify where in space it is pointing. You can call the *SetPosition()* function to set the *x,y,z* coordinates of the emitter and call the *SetOrientation()* function to define the direction in which the emitter is pointing.

```
    // Place the emitter at the origin
    v3d.x = (float)x;
    v3d.y = (float)y;
    v3d.z = (float)z;
    lpCE->SetPosition(&v3d);

    // Point the emitter along the Z axis -
    // into the computer screen.
    v3d.x = 0.0f;
    v3d.y = 0.0f;
    v3d.z = 1.0f;
    lpCE->SetOrientation(&v3d);
}
```



Similarly, let's position the direct listener object in terms of the 3D world coordinates. The direct listener has three properties: its *x,y,z* position in the 3D world, the direction in which the listener is facing, and the up direction of the listener. Notice that the up vector is always perpendicular to the orientation vector. You can use the *SetPosition()* member function to set the *x,y,z* position of the listener. You can use the *SetOrientation()* function to set both the orientation and up vectors.

```
HRESULT CRsxSampleView::SetListenerPosition(int x, int y, int z)
{
    RSXVECTOR3D v3d;
    RSXVECTOR3D v3dOrient;
    RSXVECTOR3D v3dUpOrient;

    // Set the DirectListener's position
    v3d.x = (float)x;
    v3d.y = (float)y;
    v3d.z = (float)z;
    m_lpDL->SetPosition(&v3d);

    // Listener orientation settings
    // This vector is the direction the listener is facing
    v3dOrient.x = 0.0f;
    v3dOrient.y = 0.0f;
    v3dOrient.z = 1.0f;

    // "up" vector - This vector points to which direction is up for the
    // listener - it can not be parallel to the listener orientation vector
    v3dUpOrient.x = 0.0f;
    v3dUpOrient.y = 1.0f;
    v3dUpOrient.z = 0.0f;

    // Set the orientation of the listener
    m_lpDL->SetOrientation(&v3dOrient, &v3dUpOrient);
    return 0;
}
```



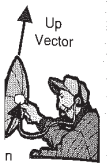
By now you've positioned the listener and emitter as if they were objects in the real world. RSX 3D uses this information to calculate the correct intensity for the left and right speakers in order to deliver a more realistic listening experience. As you move the listener and emitter objects around (by changing their position or orientation), RSX 3D automatically recalculates the correct output for both speakers.

From a 3D graphics programmer's point of view, you only need to attach an RSX 3D sound object to your current 3D graphics objects and just move it around as part of the 3D graphics object. In turn RSX 3D figures out the audio output based on the position of this object.

13.8 Adding Special Sound Effects with RSX 3D

13.8.1 The Doppler Effect

Just in case the Doppler phenomenon is new to you, the Doppler effect is the apparent change in a sound when there is relative motion between the emitter and the listener. For example, as an airplane travels toward a



listener, sound waves are compressed, effectively increasing the pitch. As the airplane travels away from the listener, the sound waves are rarefied, correspondingly decreasing the pitch. In both cases, the listener "hears" sound at a different pitch than what the emitter produced.

To enable the Doppler effect with RSX 3D, you only need to assure that the `RSXEMITTERDESC_NODOPPLER` flag is *not* set when you initialize the emitter.

13.8.2 The Reverberation Effect

Just in case reverberation is new to you, reverberation is the "slight echo" effect heard when sounds are generated in enclosed areas (from small chambers to wide canyons). Sound waves travel directly from the sound source to our ears. But in enclosed areas these waves also bounce off the surrounding walls and return to our ears many times. Reverberation is the collective effect of these indirect sound waves.

To enable the reverberation effect with RSX 3D, make sure that the `RSXEMITTERDESC_NOREVERB` flag is not set when you initialize the emitter. You then use the *SetReverb()* function to set the reverberation model parameters.

RSX 3D uses two parameters to define reverberation: decay time and intensity. The *decay time* models reverberation decay (in seconds), and the intensity models sound absorption.

```
// Must first get a pointer to an IID_RSX2 object to use the
// SetReverb() function.
HRESULT hr = m_lpUnk->QueryInterface(IID_IRSX2, (void**)&m_lpRSX);
if (FAILED(hr) || !m_lpRSX) {
    AfxMessageBox("Error getting IRSX2 interface\n");
    return 0;
}

// Now set the reverb model
RSXREVERBMODEL rsxRvb;
rszRvb.cbSize = sizeof(rsxRvb);
rsxRvb.bUseReverb = TRUE;
rsxRvb.fDecayTime = 1.5f;
rsxRvb.fIntensity = 0.1f;

m_lpRSX->SetReverb(&rsxRvb);
m_lpRSX->Release();
```

Common reverberation parameters:

Room Type	Decay	Intensity
ROOM	0.5	0.2
CHAMBER	1.0	0.2
STAGE	1.5	0.2
HALL	2.0	0.2
PLATE	2.5	0.2

13.9 Audio Streaming in RSX 3D

RSX 3D supports two more objects for audio streaming—streaming emitter and streaming listener—where the application can examine or modify the output of the emitter on its way to the listener. *Streaming emitters* are great if you dynamically generate audio input (instead of reading a file), or if you want to stream audio from a network or want to add additional effects to the data before handing it to RSX 3D. Streaming listeners are useful for mixing RSX 3D output with other audio output or writing the data to a file instead of to the audio device.

Audio streaming with RSX 3D also provides for callback mechanisms and multiple stream synchronization.

Since the streaming sound model is exactly the same as we've used for 3D sounds earlier, we prefer that you refer to the RSX 3D documentation for more details.

WHAT HAVE YOU LEARNED?

This is a good time to run the RSX 3D samples for this chapter on the CD to get the most out of this chapter.

In this chapter, you learned how easy it is to play and mix multiple audio files with RSX 3D. You were then introduced to the RSX 3D audio environment, where you learned how to set up the position and orientation of the audio objects and how to change their sound characteristics with the Doppler effect, attenuation, and reverberation. Finally, we briefly looked at the streaming objects supported by RSX 3D.

WE'D L
BROWN

PART V

Welcome to the Third Dimension

WE'D LIKE TO EXTEND AN ACKNOWLEDGEMENT TO JACKIE COLLUM, SUSAN DULIS-RINNE, SALLY BROWN (NOT RELATED TO CHARLIE BROWN), JEANNETTE MADDOX, MONICA PARDY, CHARING RIOLO WITHOUT WHOM THE WORLD WOULD NOT GO AROUND.

Chapter 14 **An Introduction to Direct3D**

- Understand Direct3D's target
- Look at Direct3D architecture and modes
- Use Direct3D to draw a simple triangle with default states

Chapter 15 **Embellishing Our Triangle with Backgrounds, Shading, and Textures**

- Add bells and whistles to the simple triangle including shading, texture mapping and Z-Buffering
- Repaint the background with Direct3D

Chapter 16 **Understanding and Enhancing Direct3D Performance**

- Measure performance of the simple triangle samples
- Use Ramp model driver to get better performance
- Measure improvements

Chapter 17 **Mixing 3D with Sprites, Backgrounds, and Videos**

- Mix 3D objects on top of 2D background
- Mix 3D objects on top of 2D sprites and video
- Use video as a texture map source

Part V deals with 3D graphics for Windows 95. We'll start in Chapter 14 with a short contextual background to 3D on the PC. That should fill you in on how 3D evolved on the PC. Then we'll dive into an overview of Microsoft's Direct3D architecture.

Next we will get you started with Direct3D programming. There is a lot to be learned before you can see results with Direct3D, so Chapter 14 has simple ambitions—to show you how to render a single triangle with Direct3D.

Once you have learned how to render the simplest triangle, you will be in a position to understand how to enable various features in Direct3D. Chapter 15 shows you how to access features such as coloring, shading, texture mapping, Z-Buffering, and repainting backgrounds. Chapters 14 and 15 show you how to get your code running; they do not worry about performance.

In Chapter 16, we return to our performance-oriented angle of rendering. First we measure the performance of the code from the previous chapters. Then we focus on the high-performance rendering path in Direct3D—the Ramp model. The Ramp model offers a significant performance boost, but the model is not straightforward. That is why we deliberately delayed introducing this performance option until after we described the basics of Direct3D rendering.

Once you know how to render high-performance 3D triangles, how about mixing in 2D graphics and video, that is, mixing in the output from the previous parts of the book? We have dedicated Chapter 17 to mixing. In keeping with the previous parts, we use the list management features of RDX to mix sprites and video objects. All the Direct3D code in Chapters 14 through 17 is based on the Direct3D ExecuteBuffer model that was released as part of Microsoft's DirectX 3.0 SDK. The ExecuteBuffer API model is hard to debug, so Microsoft is releasing a new API model (the DrawPrimitive API model), with Version 5.0 of DirectX. Version 5.0 will be released in 1998 along with Windows 98.

Why didn't we use the upcoming API? The DrawPrimitive API was still under development when we wrote the book. We decided to present you with the latest information possible. We have also "printed" this chapter in electronic form on our companion CD. This CD-based chapter will show you how to get going quickly with DrawPrimitives. The performance of DrawPrimitives will continue to improve, and we recommend that you perform your own measurements of the released version.

CHAPTER 14

An Introduction to Direct3D

WHY READ THIS CHAPTER?

We'll start this chapter with a short contextual background on 3D on the PC, and then we'll jump to an overview of Microsoft's Direct3D. Next we'll get to the main purpose of this chapter: giving you the bare bones minimum information to render a triangle using Direct3D.

By the time you have worked through this chapter, you will

- understand the problem space that Direct3D is targeted at,
- get a glimpse of the architecture of Direct3D and its different modes, and
- see how Direct3D works with DirectDraw,
- learn how to get access to 3D functionality and 3D devices,
- learn how to connect DirectDraw's surfaces and palettes to equivalent Direct3D objects, and
- learn how to use execute buffers and viewports to render a triangle using Direct3D.

PART V

14.1 Some Background on 3D on the PC

Standards for 3D (such as OpenGL and PHIGS[1]) were developed on workstations and provide powerful capabilities for 3D application developers. But these rich 3D libraries on the PC offered unacceptably poor performance when they were implemented. Developers using 3D on the PC relied

1. PHIGS stands for Programmer's Hierarchical Interactive Graphics Systems.

heavily on high-end graphics accelerators to deliver acceptable applications. These expensive 3D graphics solutions were targeted to serious users (Computer-Aided Design, CAD, for example). As a result, 3D on a PC was out of reach for the casual user.

Then enterprising game software developers invented creative techniques for reducing the computational cost of 3D (primarily by constraining the 3D models). These approximated 3D solutions still provided a compelling illusion of 3D and triggered a wave of excitement for the PC as a platform capable of delivering 3D.

Prominent vendors introduced general-purpose 3D solutions tailored specifically for the PC, including Reality Labs by Rendermorphics, BRender by Argonaut, RenderWare by Criterion, and 3DR by Intel. These general-purpose 3D libraries were not as fast as in-house solutions tailored for application-specific needs, but they were fast enough to work with undemanding 3D applications. They were also designed to use hardware accelerators when available.

Encouraged by the emergence of 3D libraries and applications, graphics vendors started building low-cost 3D hardware accelerators. Unfortunately the multitude of software solutions did not offer graphics vendors a stable target to deliver cost-reduced accelerators. Similarly, because of the numerous variations in hardware acceleration features, developers had to customize their products to each individual accelerator.

In an attempt to move toward a ubiquitous 3D solution, Microsoft started work on Direct3D, intended as an interface to 3D hardware devices. Since the feature sets of the hardware offerings differed, Microsoft realized the need for software emulation to provide developers with a minimum baseline of functionality. In 1995 Microsoft bought Rendermorphics to integrate Reality Labs into their universal 3D solution. In working toward a universal 3D solution, Microsoft aimed at providing hardware vendors with a single driver model at which they could target their accelerators.

The initial response to Microsoft's software emulator was a consistent demand for more performance. Microsoft responded by providing Direct3D with two modes—Retained mode and Immediate mode—offering different feature and performance capabilities. In addition, Microsoft provided two different implementations of the software emulation pipeline—RGB and Mono. The various combinations of modes and drivers offer a variety of API abstractions, feature sets, and quality and performance levels.

So 3D on the PC has become a reality, although it is still in its fledgling state; whereas video, audio, and 2D have had a few years and several iterations to mature. Therefore we will continue to see evolutions in performance, quality, functionality, and API abstractions in future offerings for 3D. Nonetheless, Microsoft's Direct3D has established itself as the foundation for further iterations.

14.2 Introduction to Direct3D

Figure 14-1 shows the current display architecture available under Windows 95. In this chapter we are concerned with the interfaces within the ellipse outlining the Direct3D boxes.

Direct3D is part of Microsoft's DirectX SDK. To application developers, Direct3D provides APIs and services for 3D manipulations. To hardware vendors Direct3D provides a single driver model to enable hardware acceleration. Most significantly, Direct3D guarantees 3D functionality to software developers with a software-based emulation layer. Hardware vendors can accelerate those features that they feel fit their price/performance budget.

Direct3D is closely integrated with DirectDraw. Direct3D makes extensive use of the DirectDraw surface model to access hardware acceleration features such as Bltters and Page Flippers. This integration with DirectDraw makes it possible for Direct3D to use advanced features such as video textures and 2D overlays.

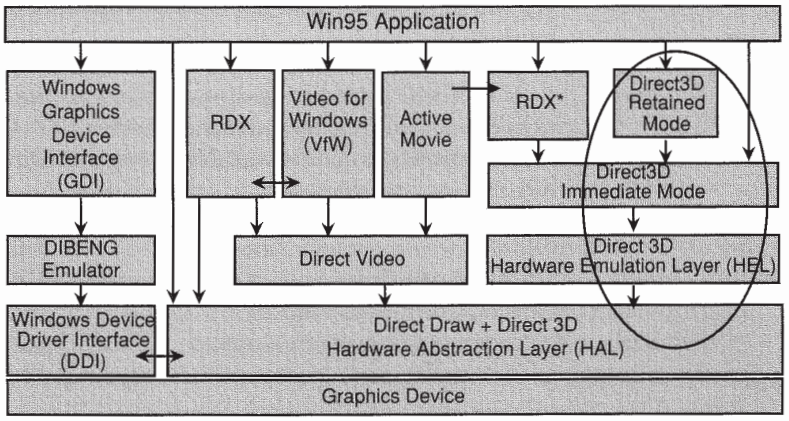


FIGURE 14-1 Display architecture under Windows 95.

Direct3D offers 3D features with two distinctively different flavors: Retained mode and Immediate mode.

- *Retained mode* offers high-level abstraction of 3D objects and manipulations. It has a sophisticated geometry engine that allows entire scenes to be manipulated with high-level API calls. But the functionality comes at a performance cost.
- *Immediate mode* offers a very thin layer of software functionality with high performance. It also offers direct access to hardware acceleration features. But Immediate mode does not have a geometry engine, and object transformations must be computed by the application itself.

The two Direct3D modes offer different levels of API abstractions. In addition, multiple implementations of Direct3D drivers can be installed on a system to offer different combinations of features, levels of performance, and quality.

The DirectX SDK ships with two implementations: RGB and Mono. The RGB driver offers truer color quality at a performance cost. The Mono driver makes color approximations and delivers higher performance at a cost in quality. In addition, hardware vendors make available additional Direct3D drivers to offer acceleration features.

14.2.1 A Taste of Direct3D's Retained Mode

Direct3D's Retained mode API is designed for managing entire 3D scenes. In this mode you can load predefined 3D objects from files and manipulate these objects without having to explicitly perform any matrix computations. When you integrate Retained mode with Direct3D authoring tools, you can generate entire 3D applications with minimal explicit programming effort.

Retained mode provides object abstractions and methods on these objects to free you from creating and managing the details of internal object databases. Some of the objects available through the Direct3D Retained mode API are listed in Table 14-1.

Although you can manipulate entire scenes, the Retained mode does not as yet offer compelling performance. So our use of Direct3D will focus on Direct3D's Immediate mode.

TABLE 14-1 Objects Abstracted by Direct3D's Retained Mode Interface

Object	Description
Direct3DRMDevice	Renderer destination
Direct3DRMFace	Represents a single polygon
Direct3DRMMesh	Grouping of polygonal faces and vertices
Direct3DRMMeshBuilder	Build vertices and faces into a mesh
Direct3DRMFrame	Positions objects within a scene
Direct3DRMLight	Five options of lights to illuminate objects in a scene
Direct3DRMMaterial	Properties describing how faces reflect light
Direct3DRMShadow	Define shadows on objects
Direct3DRMTexture	Rectangular image to be rendered onto polygons
Direct3DRMViewport	Define how a 3D scene is rendered into a 2D window
Direct3DRMPickedArray	Choose an object corresponding to a 2D point
Direct3DRMVisual	Placeholder for anything that can be rendered in a scene
Direct3DRMAnimation	Series of transformations that can be applied to a scene
Direct3DRMAnimationSet	Allows animation objects to be grouped together

14.2.2 Direct3D's Immediate Mode

Direct3D's Immediate mode is targeted for developers of high-performance 3D applications for the Microsoft Windows operating system. In designing this mode, Microsoft expected that developers using Immediate mode would be experienced in high-performance programming issues as well as 3D graphics.

Table 14-2 lists the objects (and methods) offered by the Direct3D Immediate mode API. Immediate mode offers direct access to the rendering pipeline. At its essence, Immediate mode is a device-independent way for applications to access low-level hardware acceleration. Direct3D's Retained mode is built on top of Immediate mode.

Low-level hardware accelerators are typically designed to accelerate the pixel-rendering stage, and they rely on the host CPU to compute geometry or lighting factors. But several factors from the geometry and lighting stages can affect rendering results, and some graphics accelerators offer advanced capabilities to render these influences. The Immediate mode pipeline contains objects such as *viewports, materials,* and *lights* to enable even lighting and geometry stages to be accelerated.

TABLE 14-2 Objects Offered via the Direct3D Immediate Mode Interface

Object Type	Description
Device	Hardware device (equivalent to DirectDraw surface)
ExecuteBuffer	List of vertex data and render instructions
Texture	DirectDraw surface containing a texture map image
Light	Light sources
Matrix	Four-by-four homogeneous transformation matrix
Material	Coloring options, such as color and texture
Viewport	Screen region to draw to

14.2.3 Before You Get Overly Excited

As we mentioned before, today's PCs are not yet capable of manipulating entire scenes at compelling performance levels. Hence, you'll have to approximate the complexity of your 3D models so that you can obtain an illusion of 3D at a low performance cost. Following are some examples of approximations:

- Using only rectangular walls within a building reduces geometry calculations to entire wall faces, even though you may want to subdivide the wall for better textural and rendering quality.
- Similarly, maintaining simple angles of intersection (30°, 45°, 60°, and 90°) among the walls can reduce the cost of computing lighting values and marking hidden surfaces.
- Using multiple versions of textures ("pre-lit") to simulate various lighting shades can eliminate the cost of rendering using other more costly lighting options.

Approximations like these are very application specific. The gains are only obtained when the application handles its own geometry and lighting calculations, so Direct3D's Immediate mode was designed to be used by this type of application. If you want high-performance 3D results using Direct3D's Immediate mode, you will need to have (or develop) your own geometry and lighting modules in your application.

Given that many users of Direct3D's Immediate mode are capable of developing their own geometry and lighting modules, the API and objects of the Immediate mode interface have been designed for developers with an advanced knowledge of 3D. If you are new to 3D, we strongly recommend that you read up on material about 3D geometry and lighting before attempting to develop any significant applications using Direct3D's Immediate mode.

14.3 Inside Direct3D

Before starting to use Direct3D, let's look at how Direct3D interacts with DirectDraw and sneak a peek inside Direct3D's architecture.

14.3.1 Direct3D and DirectDraw

Direct3D is very closely connected with DirectDraw. So much so that Direct3D is almost an extension of DirectDraw. This close connection is deliberate, because it lets you incorporate cool features such as texture mapping with video rendered into DirectDraw surfaces, or to overlay 3D scenes on 2D compositions.

There are four points of connection between Direct3D and DirectDraw involve interface objects and buffers.

- *IDirect3D.* The primary interface to Direct3D, IDirect3D is derived by creating an IDirectDraw object and querying (via *QueryInterface*) for a IID_IDirect3D interface.
- *IDirect3DDevice.* This interface gives you access to low-level Direct3D rendering functions. IDirect3DDevice is similar to using IDirectDrawSurface in DirectDraw to access low-level 2D functions. An IDirect3DDevice is "created" by creating an IDirectDrawSurface and querying for a 3D device GUID. The 3D device will render pixels to the 2D surface. In addition, you can use all standard DirectDraw functions on the 2D surface.
- *IDirect3DTexture.* This interface manages textures in Direct3D. IDirect3DTexture, like IDirect3DDevice, is an extension of IDirectDrawSurface and is "created" by creating a IDirectDrawSurface and querying for an IID_IDirect3DTexture interface. The 3D device will use the surface as a source texture during texture-mapped rendering. In addition, you can access all normal DirectDraw surface functions on the 2D surface.
- *Z-Buffers.* In Direct3D Z-Buffers are DirectDraw surfaces created with a DDSCAPS_ZBUFFER flag. The Z-Buffer therefore is easily visible by all modules. With Z-Buffers you can use normal 2D functions for carrying out simple operations on the Z-Buffer (such as clear).

14.3.2 Direct3D Rendering Engine

Figure 14-2 looks inside the Rendering engine of Direct3D. The *rendering engine* consists of three modules: the Transform module, the Lighting module, and the Raster module.

- The *Transform module* converts input vertices from model coordinates to render coordinates via a transform matrix created from world, view, and projection matrices. The Transform module also culls objects to fit within a specified viewport.

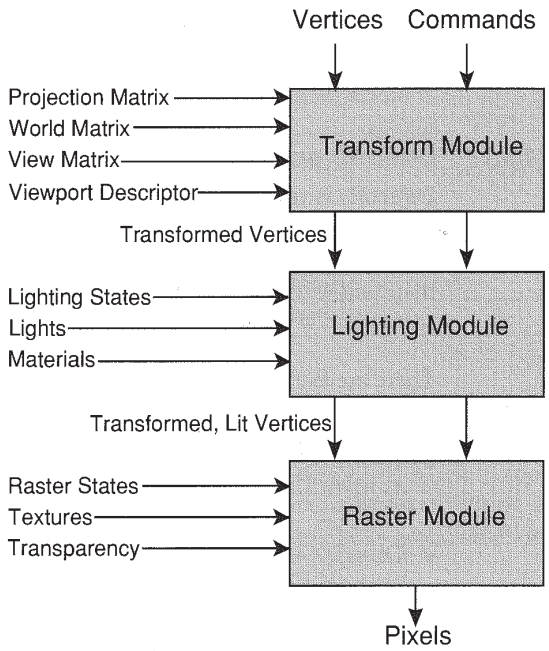


FIGURE 14-2 Direct3D Rendering engine.

- The *Lighting module* supports ambient, point, spotlight, or directional light sources and adds color information to the vertices provided by the Transform module.
- Based on raster options, such as wire-frame, solid-fill, or texture-map, the *Raster module* renders pixels conforming to the vertices and color values passed on.

All three modules are replaceable. Direct3D comes with one transformation module but with a choice of two lighting and two rasterization modules (RGB or mono). Graphics hardware vendors can provide additional replacement modules that support their 3D accelerators.

14.4 Revving Up Direct3D

Roll up your sleeves, it's coding time again. Yeeha! Over the course of working with Direct3D we will come across the objects listed in Table 14-3. We will describe each one as we get to it.

TABLE 14-3 Direct3D Objects Used in This Chapter

Object	Description
IDirect3D	Primary interface to Direct3D
IDirect3DDevice	3D device (equivalent to DirectDraw surface)
IDirect3DViewport	Screen region to draw to
IDirect3DExecuteBuffer	List of vertex data and render instructions

14.4.1 The Starting Point: IDirect3D Object

IDirect3D, as we mentioned above, is the primary interface for Direct3D. Objects such as lights, materials, and viewports are created using the IDirect3D interface. IDirect3D also has functions to enumerate (or find) 3D devices, since there can be multiple Direct3D device modules installed on a system.

Let's initialize DirectDraw and get access to Direct3D:

```
BOOL CSharedHardware::Init(HWND hWnd) {
    LPDIRECT3D    pD3D;

    // create a DirectDraw instance
    DirectDrawCreate(NULL, &m_pDDraw, NULL);

    // "create" D3D object
    m_pDDraw->QueryInterface(IID_IDirect3D, (void *)&pD3D);

    // remember to set DDraw cooperative level
    m_pDDraw->SetCooperativeLevel(hWnd, DDSCL_NORMAL);

    // assign into member variable and return
    m_pD3D = pD3D;
    return TRUE;
}
```

Invoking *IUnknown::QueryInterface()* on the IDirectDraw object with the predefined GUID *IID_IDirect3D* returns a pointer to an IDirect3D object in the second parameter.

QueryInterface() does not create a new object; instead it provides a second interface to the DirectDraw object and increments its reference count. You must call both *IDirectDraw::Release()* and *IDirect3D::Release()* to fully release the object. If you only release the IDirect3D interface and then query for it again, the original IDirect3D state might be restored.

14.4.2 Enumerating IDirect3DDevices

We have access to the first level of Direct3D. You would think that the next step would be to ask for access to an IDirect3DDevice. But wait! To ask for access to a new interface, we've got to know the new interface's GUID. So how do we get the GUID of an IDirect3DDevice?

Direct3D was built to allow multiple 3D devices to be installed in a system. We could get the GUID from our vendor of choice and hard code it into our application. An alternate approach would be to use *IDirect3D::EnumDevices* to search among installed devices and pick a device of our choosing.

```
extern "C" static HRESULT WINAPI enumDeviceFunc(
    LPGUID lpGuid, LPSTR lpDeviceDescription, LPSTR lpDeviceName,
    LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc, LPVOID lpCookie
);
```

IDirect3D::EnumDevices will call a callback function of this form for each driver installed in the system. Note that "C" calling convention is used. The parameters passed to the callback function are

LPGUID lpGuid	Pointer to the GUID for this driver
LPSTR lpDeviceDescription	String describing the driver. (For example: "Microsoft Direct3D Mono (Ramp) Software Emulation")
LPSTR lpDeviceName	String name of the driver. (For example: "Ramp Emulation")
LPD3DDEVICEDESC lpHWDesc	If this descriptor is valid, then driver is hardware based.
LPD3DDEVICEDESC lpHELDesc	If this descriptor is valid, then driver is software emulation.
LPVOID lpCookie	LPVOID sized data passed on from the main application.

```
BOOL CSharedHardware::InitDirect3D(DWORD dwCookie)
{
    // enum drivers and pick one
    // Ask D3D to call our "C" callback
    m_pD3D->EnumDevices(enumDeviceFunc, (LPVOID)dwCookie);

    // return
    return TRUE;
}
```

Invoke *IDirect3D::EnumDevices()* with the address of our callback function.
dwCookie, an LPVOID sized object, can be anything we want. It will be passed on unchanged to our callback function.

IDirect3D::EnumDevices will call our callback function once for each driver installed in the system, giving us the driver's GUID, a couple of text strings identifying the driver, and two D3DDEVICEDESC descriptors to tell us about the capabilities of each driver.

Only one of the two D3DDEVICEDESC descriptors is valid. Browsing through Direct3D's sample code, we found that the prescribed method of checking the validity of a D3DDEVICEDESC descriptor is to check whether the dcmColorModel field is set to a valid value (currently it can be either D3DCOLOR_MONO or D3DCOLOR_RGB).

We get information for a single 3D device on each call of our callback. If this device matches our selection criterion, we can tell Direct3D to stop calling us by returning D3DENUMRET_CANCEL. Otherwise, our callback is supposed to return D3DENUMRET_OK—and Direct3D will continue to call us for any remaining un-enumerated choices.

We designed our callback function to choose the first driver that matched an input criterion. Our input criterion can be:

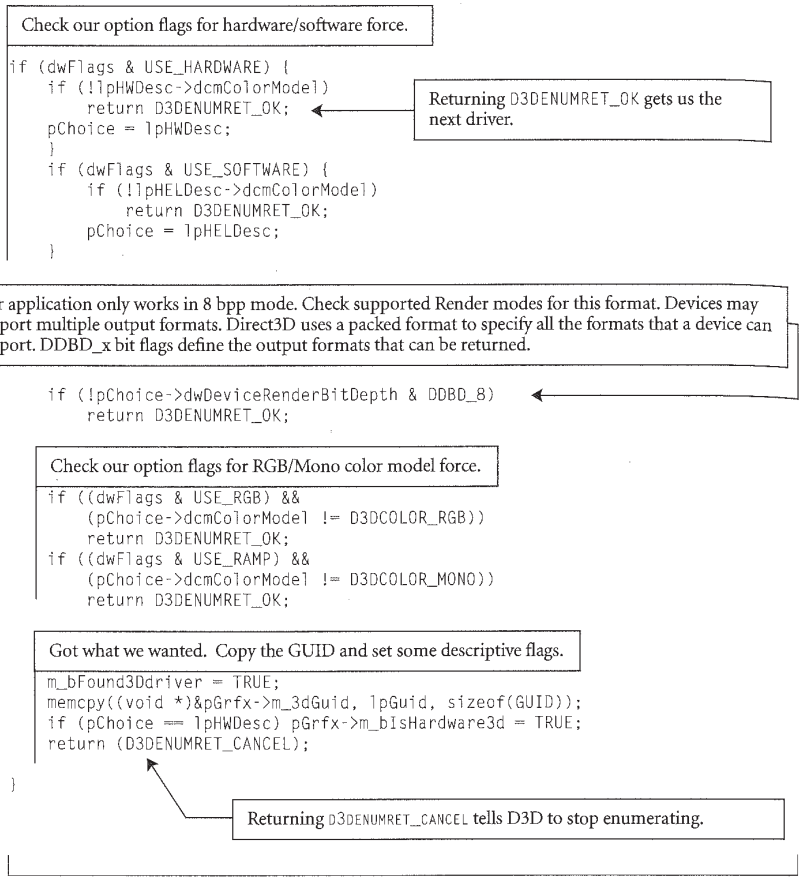
USE_HARDWARE	Reject any software emulation drivers. Specifying both USE_HARDWARE and USE_SOFTWARE is illegal; neither is ok, the first one will be chosen.*
USE_SOFTWARE	Reject any hardware drivers. Specifying both USE_HARDWARE and USE_SOFTWARE is illegal; neither is ok, the first one will be chosen.*
USE_RGB	Use the higher-quality RGB model. Specifying both USE_RGB and USE_RAMP is illegal; neither is ok, the first one will be chosen.*
USE_RAMP	Use the lower-quality, higher-performance Mono/Ramp model. Specifying both USE_RGB and USE_RAMP is illegal; neither is ok, the first one will be chosen.*
USE_ANY	Use the first 3D driver enumerated by Direct3D

* If you don't specify a choice, then the choice will be made on a first-come basis. If you specify a choice, then your choice will be honored.

Here is the code for our callback function:

```
extern "C" static HRESULT WINAPI enumDeviceFunc(
    LPGUID lpGuid, LPSTR lpDeviceDescription, LPSTR lpDeviceName,
    LPD3DDEVICEDESC lpHWDesc, LPD3DDEVICEDESC lpHELDesc, LPVOID lpCookie
)
{
    DWORD dwFlags = (DWORD)lpCookie;
    CSharedHardware *pGrfx  = gpAppWide->m_pGrfxCard;

    Pick the valid driver from the two device descriptors.

    LPD3DDEVICEDESC pChoice = lpHWDesc;
    if (!pChoice->dcmColorModel)
        pChoice = lpHELDesc;
```

The method to check whether a D3DDEVICEDESC is valid—as approved in Direct3D sample code—is to check dcmColorModel for a valid value.



When control returns from Direct3D to the original function that invoked *IDirect3D::EnumDevices*, our callback function would have copied a GUID for a Direct3D device that matched our specification (if there was one). We can now use this GUID to query for a IDirect3Ddevice object.

14.4.3 Creating an IDirect3DDevice

Now we're really getting down! An IDirect3DDevice interface provides low-level access to Direct3D rendering functions. IDirect3DDevice is not an object in its own right; it is an extension to an IDirectDrawSurface object. To get an IDirect3DDevice object, we've got to first create an IDirectDrawSurface and then "extend" the surface by querying for 3D capabilities.

To extend an IDirectDrawSurface into an IDirect3DDevice, the surface needs to have been created using DDSCAPS_3DDEVICE set in the surface caps (*ddsCaps.dwCaps*) field.

Let's create a suitable IDirectDrawSurface. We don't want to see the flicker that results from compositing directly onto the display screen, so we're using an Offscreen surface (although tests showed that we could successfully extend the Primary surface for 3D). The code for creating an IDirectDrawSurface is pretty much the same as the code we used in Chapter 5, except for the addition of the DDSCAPS_3DDEVICE flag:

```
CSurfaceSysMem::Init(CWnd *pcWnd, LPDIRECTDRAW2 pDDraw)
{
    // get size of client to create similar off.screen window
    RECT    rWin;
    pcWnd->GetClientRect(&rWin);
    m_dwWidth = (dword)(rWin.right - rWin.left);
    m_dwHeight = (dword)(rWin.bottom - rWin.top);

    // init surface descriptor and create offscreen surf
    m_SurfDesc.dwHeight = m_dwHeight & (~0x03);
    m_SurfDesc.dwWidth = m_dwWidth & (~0x03);

    // specify desired 8bpp color format
    m_SurfDesc.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
    m_SurfDesc.ddpfPixelFormat.dwRGBBitCount = 8;
    m_SurfDesc.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8 | DDPF_RGB;

    // ask for offScreenSurface, in system memory, with 3d capabilities
    m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
    m_SurfDesc.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;
    m_SurfDesc.ddsCaps.dwCaps |= DDSCAPS_3DDEVICE;

    // specify which fields in SurfDesc are valid
    m_SurfDesc.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT | DDSD_PIXELFORMAT;

    // create the surface
    pDDraw->CreateSurface(&m_SurfDesc, &m_pSurfFns, NULL);
    return TRUE;
}
```

Add the DDSCAPS_3DDEVICE flag to tell DirectDraw that we will extend this surface for 3D.

Even though we've created a suitable DirectDraw surface, we are not yet ready to extend it for 3D. When using Direct3D to write into a palletized surface, we must attach a palette to the surface before extending the surface for 3D.

14.4.4 Preparing a DirectDraw Palette

Following is the code for creating a palette using DirectDraw's Palette functions. We've initialized the new palette to the system palette and attached it to both the Primary and the Offscreen surfaces.

The Direct3D RGB driver operates in a high-quality RGB color model. It must then reduce the colors in the scene to appropriate palette entries. Direct3D needs to be able to select these palette entries. So we've "prepared" the palette for Direct3D by setting flags that tell Direct3D which values it can modify.

```
BOOL CSharedHardware::InitPalette(void)
{
    // Get the current system palette.
    PALETTEENTRY ppeSysPal[256];
    HDC hdc = GetDC(NULL);
    GetSystemPaletteEntries(hdc, 0, (1 << 8), ppeSysPal);
    ReleaseDC(NULL, hdc);

    int i;
    for (i = 0; i < 10; i++)
        ppeSysPal[i].peFlags = D3DPAL_READONLY;
    for (i = 10; i < 246; i++)
        ppeSysPal[i].peFlags = D3DPAL_FREE | PC_RESERVED ;
    for (i = 246; i < 256; i++)
        ppeSysPal[i].peFlags = D3DPAL_READONLY;

    // create palette and init with above values
    m_pDDraw->CreatePalette(DDPCAPS_8BIT | DDPCAPS_INITIALIZE,
                            ppeSysPal, &m_pPalette, NULL);
    return TRUE;
}

BOOL CSharedHardware::SetPalette(LPDIRECTDRAWSURFACE2 pSurfFns)
{
    // set created palette on specified surface
    pSurfFns->SetPalette(m_pPalette);
    return TRUE;
}
```

Allow D3D to change middle entries. For windowed case, preserve top ten and bottom ten colors. In Full Screen mode we could allow all but the top and bottom color to be changed.

Create a DirectDraw palette. A pointer to an IDirectDrawPalette object is returned in the third parameter.

14.4.5 Extending the Surface for 3D

Now we're ready to extend the DirectDraw surface and get an IDirect3DDevice object:

```
BOOL CSurface3d::Init(CWnd *pcWnd, LPDIRECTDRAWSURFACE2 p2dFns)
{
    p2dFns->QueryInterface(gpAppWide->m_pGrfxCard->m_3dGuid, &m_p3dFns);
```

We're invoking *IUnknown::QueryInterface()* on the IDirectDrawSurface object with the GUID that we chose earlier in our *EnumDevices* callback function. We get back a pointer to the IDirect3DDevice object in the second parameter.

Get the 2D surface descriptors and copy some info about 2D surface.

```
    DDSURFACEDESC ddsdTmp;
    memset(&ddsdTmp, 0, sizeof(DDSURFACEDESC));
    ddsdTmp.dwSize = sizeof(DDSURFACEDESC);
    err = p2dFns->GetSurfaceDesc(&ddsdTmp);        // get surface descriptor
    m_dwWidth = ddsdTmp.dwWidth;                   // get width of 2d surface
    m_dwHeight = ddsdTmp.dwHeight;                 // get height of 2d surface
    if (ddsdTmp.ddsCaps.dwCaps & DDSCAPS_VIDEOMEMORY) m_bIsVidMem = TRUE;
    m_p2dFns = p2dFns;                              // remember 2d surface fns

    return TRUE;
}
```

Voilà! Pardon my French, but I am happy to announce that the IDirectDrawSurface has now been extended to allow for 3D capabilities. Also allow me to point out again that *QueryInterface()* does not create a new object. Instead, it provides a second interface in addition to the original IDirectDrawSurface object and increments its reference count. You must call both *IDirectDrawSurface::Release()* and *IDirect3DDevice::Release()* to fully release the object.

Note that only this surface has been "extended" for 3D capabilities. As such we could say that this surface is a "3D surface." If you browse our code for this chapter on the CD, you will notice that we use *Surface3D* to refer to IDirect3DDevice, because as we said before, it's only this surface that has been 3D enabled.

Just as with 2D surfaces, 3DDevices are described by a descriptor, `D3DDEVICEDESC`. You can use *IDirect3DDevice::GetCaps* to get the descriptor of the device. In this chapter we are mainly concerned with getting a triangle rendered through Direct3D without adding any bells and whistles—yet. So we don't really need to look at the device capabilities and the `D3DDEVICEDESC` structure at this point. We'll leave that for our later chapter on accelerating Direct3D, Chapter 15.

14.4.6 Mapping from a 3D Model to the 2D Surface Using Viewports

We've got our 3D surface (3DDevice), and pretty soon we'll want to render some 3D objects. 3D objects are represented with 3D coordinates in a 3D model. We will have to tell Direct3D how to project 3D objects onto a 2D screen. Direct3D provides an IDirect3DViewport object to control this mapping.

A *viewport* defines a visible 3D volume and the projection of this 3D volume onto a 2D screen area. For perspective viewing, the visible 3D volume is a portion of a pyramid between a front clipping plane and a back clipping plane. For orthographic viewing, the visible 3D volume is cuboid.

For perspective projection, the viewing position is at the tip of the pyramid as in Figure 14-3. The *z*-axis runs from the tip of the pyramid to the center of the pyramid's base. The front clipping plane is at a distance *P*, the back clipping plane is at a distance *Q* from the front clipping plane. The height of the front clipping plane is 2*H*, and it defines the field of view.

With Direct3D's Retained mode, you could use *IDirect3DRMViewport ::SetFront(), ::SetBack()*, and *::SetField()* functions to set the values of *P, Q*, and *H*, respectively. But with Direct3D's Immediate mode you've got to compute equivalent values for *P, Q*, and *H* and fill these values into a `D3DVIEWPORT` structure.

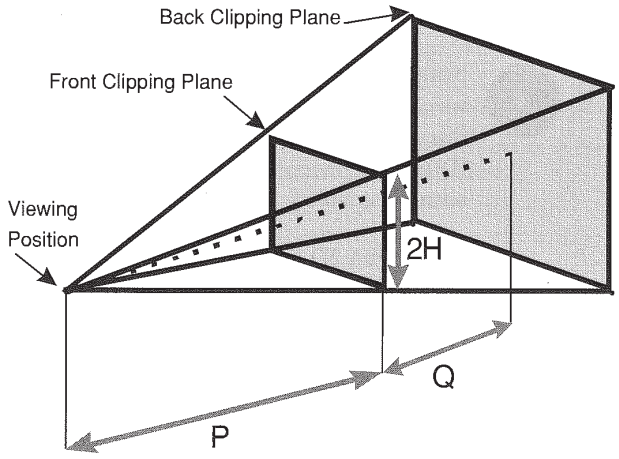


FIGURE 14-3 Using a viewport for perspective projection.

Let's take a look at the D3DVIEWPORT structure:

```
typedef struct _D3DVIEWPORT {
    DWORD    dwSize;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwWidth;
    DWORD    dwHeight;
    D3DVALUE dvScaleX;
    D3DVALUE dvScaleY;
    D3DVALUE dvMaxX;
    D3DVALUE dvMaxY;
    D3DVALUE dvMinZ;
    D3DVALUE dvMaxZ;
} D3DVIEWPORT, *LPD3DVIEWPORT;
```

Coordinates of the top-left corner of the viewport and dimensions of the viewport. These are defined relative to the top left of the device.

Scale parameters can be used to maximize the window area occupied by the 3D scene. For example, Direct3D suggests that the scale parameters be set such that the larger dimension (width or height) of the front plane fills the window.

dvMaxX, dvMaxY, dvMinZ, and dvMaxZ describe the maximum and minimum homogeneous coordinates of *x*, *y*, and *z*. Use these coordinates to describe the viewing volume.

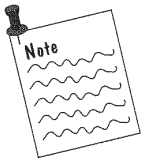
We create an IDirect3DViewport object by using *IDirect3D::CreateViewport()*, and we set the viewport's parameters using *IDirect3DViewport::SetViewport()*. Once we've created an IDirect3DViewport object we've got to associate it to our 3D surface using *IDirect3DDevice::AddViewport()*. Here's the code for doing that:

```
BOOL CSurface3d::InitViewport(lpdirect3d pD3D)
{
    pD3D->CreateViewport(&m_p3dViewport, null) ;   ◊ Use IDirect3D to create the viewport.
    m_p3dFns->AddViewport(m_p3dViewport);          ◊ Once created, attach the viewport to 3D surface.

    // Setup viewport to be equivalent to window
    D3DVIEWPORT viewData;
    memset(&viewData, 0, sizeof(D3DVIEWPORT));
    viewData.dwSize = sizeof(D3DVIEWPORT);
    viewData.dwX = 0;
    viewData.dwY = 0;
    viewData.dwWidth = m_dwWidth;
    viewData.dwHeight = m_dwHeight;
    viewData.dvScaleX = (float)1.0;
    viewData.dvScaleY = (float)1.0 ;
    viewData.dvMaxX = (float)viewData.dwWidth;
    viewData.dvMaxY = (float)viewData.dwHeight;
    m_p3dViewport->SetViewport(&viewData);

    return true;
}
```

Our purpose in this chapter is merely to render a triangle. So we've set up the simplest possible viewport, where the viewport is the same size as the drawing window, and there's no scaling and projection.

PART V

Based on our viewport parameters, the driver builds a transformation matrix to convert incoming vertices from a 3D model space to a projected 2D space.



Direct3D lets us create multiple viewports. We tell Direct3D which viewport to use only when we're rendering an object. In this way we could, if we wanted, mix objects rendered with different perspectives onto the same surface.

14.4.7 Talking to 3D Devices Through Execute Buffers

Okay, we've got our 3D device called Surface3D on the CD, and we've described our viewport. Now let's render some triangles, which brings up our next step—talking to the 3D device.

We send instructions to 3D devices in lists called *Execute Buffers*. Figure 14-4 is a picture of an Execute Buffer. Data items sent to 3D devices via Execute Buffers are usually either triangle vertices or render operations.

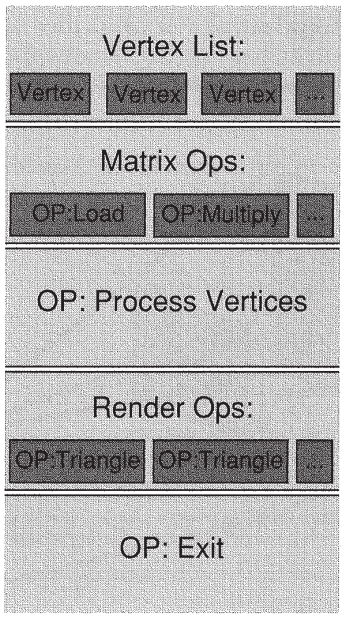


FIGURE 14-4 Sample Execute Buffer.

A vertex is typically sent using a D3DVERTEX structure. A vertex processed by the Transform module and Lighting modules is converted into a D3DTLVERTEX structure, which occupies the same data space as a D3DVERTEX structure. This process allows transformation and lighting to be performed in place. To make in-place transformation and lighting convenient, 3D devices expect vertices to precede all operations.

3D devices can be commanded to perform operations using a set of *opcodes* defined by Direct3D. We will examine the opcodes in more detail shortly (in 14.4.8). In general, there are opcodes to load data (such as matrices or textures) into device memory; opcodes to set state values in the Transformation, Lighting, or Render modules; and opcodes to process data.

IDirect3DDevice has a *CreateExecuteBuffer()* method to create an Execute Buffer object. The IDirect3DExecuteBuffer object returned is only an interface object and does not yet provide the actual buffer space into which you can insert commands.

When we're asking an IDirect3DDevice object to create an Execute Buffer, we've got to describe the buffer we'd like created. Here's the D3DEXECUTEBUFFERDESC structure used to describe our needs:

```
typedef struct _D3DExecuteBufferDesc {
    DWORD  dwSize;
    DWORD  dwFlags ;
    DWORD  dwCaps;
    DWORD  dwBufferSize;
    LPVOID lpData;
} D3DEXECUTEBUFFERDESC;
```

Hardware devices prefer their data (including vertices) to reside in video memory. Software emulators, on the other hand, prefer that their data reside in system memory. Specify the memory type in the dwCaps field based on the 3D device created. SystemMemory is the default if the field is left unspecified. The field can be

D3DDEBCAPS_SYSTEMMEMORY	The Execute Buffer data must reside in system memory.
D3DDEBCAPS_VIDEOMEMORY	The Execute Buffer data must reside in video memory.
D3DDEBCAPS_MEM	A logical OR of D3DDEBCAPS_SYSTEMMEMORY and D3DDEBCAPS_VIDEOMEMORY.

In the same lines as the DirectDraw convention, not all fields in descriptor structures are valid. dwFlags specifies which fields in the descriptor have been set. For D3DEXECUTEBUFFERDESC the flags are

D3DDEB_BUFSIZE	The dwBufferSize member is valid.
D3DDEB_CAPS	The dwCaps member is valid. [See above].
D3DDEB_LPDATA	The lpData member is valid. The 3D device returns a pointer to Execute Buffer data space in this field. Could be used to provide the driver with a pre-initialized buffer, but this doesn't always work.

Here's code that demonstrates the creation of an Execute Buffer:

```
// compute size of execute buffer needed to render one triangle
size_t sztEx =  0;
sztEx = sizeof(D3DTLVERTEX) * 3;
sztEx += sizeof(D3DINSTRUCTION)*3;
sztEx += sizeof(D3DPROCESSVERTICES);
sztEx += sizeof(D3DTRIANGLE) * 1;
```

We need to tell the device how large a buffer we will need. For our example we are requesting enough buffer space to contain commands to render one triangle. Don't worry about the actual sizes and instructions in the Execute Buffer—we'll be looking at this shortly.

```
// Describe ExecuteBuffer to be created
D3DEXECUTEBUFFERDESC debDesc;
memset(&debDesc, 0, sizeof(debDesc));
debDesc.dwSize = sizeof(debDesc);
debDesc.dwFlags = D3DDEB_BUFSIZE;
debDesc.dwBufferSize = sztEx;
```

Some 3D devices limit the size of the Execute Buffer that can be created. *CreateExecuteBuffer()* may fail if you request too large a buffer. You can find out the allowed limit by using *GetCaps()*.

```
// Create ExecuteBuffer
m_p3dFns->CreateExecuteBuffer(&debDesc, & m_pExBufFns, NULL) ;
```

Create the Execute Buffer according to description passed in first parameter. An interface object pointer (LPDIRECT3DEXECUTEBUFFER) is returned in the second parameter.

As we mentioned before, the IDirect3DExecuteBuffer object we just created is only an interface object and does not yet provide buffer space for inserting commands. We get access to usable buffer space by using the *IDirect3DExecuteBuffer::Lock* function. The lock returns a buffer pointer into which we can enter our commands. Once we finish entering our commands, we must use the ::*Unlock()* function to unlock the buffer. Here's a quick example:

Copy the triangle into the Execute Buffer space. Assume that the commands were in a preset buffer. Don't worry about the actual instructions—we'll examine these in detail shortly.

```
m_pExBufFns->Lock(&m_ExDesc);       // lock buffer
PVOID pTmp = m_ExDesc.lpData;       // get returned ptr
memcpy(pTmp, pSomeBuffer, sztEx);   // copy data over
m_pExBufFns->Unlock();              // unlock buffer
```

After entering the commands, we tell the 3D device that we've given it new instructions by invoking ::*SetExecuteData()*. At this point we're also describing to the 3D device the makeup of our Execute Buffer—where the vertices start, how many vertices there are, where the instructions start and end.

This information is provided to the 3D device using a D3DEXECUTEDATA structure. The structure is simple, and its use is demonstrated in the following code:

```
// describe make-up of recently copied execute buffer
D3DEXECUTEDATA ExecData;
memset(&ExecData, 0, sizeof(D3DEXECUTEDATA));
ExecData.dwSize = sizeof(D3DEXECUTEDATA);
ExecData.dwVertexOffset = 0;
ExecData.dwVertexCount = 3;
ExecData.dwInstructionOffset = sizeof(D3DTLVERTEX) * 3;
ExecData.dwInstructionLength = sizSomeBuffer - (sizeof(D3DTLVERTEX)*3);
pExecCmds->SetExecuteData(&ExecData);
```

After transferring an Execute Buffer to the 3D device using the Lock/Copy/Unlock sequence, we need to describe the makeup of the recently copied Execute Buffer, using *SetExecuteData()* and a D3DEXECUTEDATA structure. Here are the fields in the structure:

dwVertexOffset	Where do vertices start within the Execute Buffer?
dwVertexCount	Number of vertices in vertex list.
dwInstructionOffset	Where do instructions start within the Execute Buffer?
dwInstructionLength	Where do vertices end? This need not be the end of the buffer.
dsStatus	D3DSTATUS structure used to return the screen extents needed after vertex transformations.

OK, we've waited long enough. Let's see what operations we can perform with Direct3D.

14.4.8 Execute Operations

Operands are passed to the 3D device using a _D3DINSTRUCTION structure:

```
typedef struct _D3DINSTRUCTION {
    BYTE bOpcode;
    BYTE bSize;
    WORD wCount;
} D3DINSTRUCTION, *LPD3DINSTRUCTION;
```

The first field in the _D3DINSTRUCTION structure is the opcode. Opcodes available in Direct3D are listed in Table 14-4. With the exception of D3DOP_EXIT and D3DOP_NOP, all operations are followed by an operand. Operands are specified with a structure format unique to the operation. D3DOP_POINT, for example uses a D3DPOINT structure. The sizes of operand structures vary, and they must be entered in the bSize field. The sizes are used to advance pointers while parsing instructions in an instruction stream.

TABLE 14-4 Direct3D Execute Opcodes

Opcode	Purpose
D3DOP_TEXTURELOAD	Causes device to load a texture into device data space
D3DOP_MATRIXLOAD	Causes device to load a texture into device data space
D3DOP_MATRIXMULTIPLY	Causes matrix to multiply via the rendering pipeline
D3DOP_STATETRANSFORM	Sets value of specified transformation module state variable
D3DOP_STATELIGHT	Sets value of specified lighting module state variable
D3DOP_STATERENDER	Sets value of specified render module state variable
D3DOP_POINT	Renders a point via the renderer
D3DOP_SPAN	Spans a list of points with the same y value
D3DOP_LINE	Renders a line via the renderer
D3DOP_TRIANGLE	Renders a triangle via the renderer
D3DOP_PROCESSVERTICES	Causes vertices to be transformed, lit, and copied to device space
D3DOP_BRANCHFORWARD	Enables a branching mechanism within an Execute Buffer
D3DOP_NOP	Used for optimization to align data on QWORD boundaries
D3DOP_EXIT	Signals that the end of the list has been reached
D3DOP_SETSTATUS	Resets the status of the Execute Buffer

In a typical usage scenario, operations are quite often repeated with different parameters. For example, let's look at rendering multiple triangles. The wCount field in the _D3DINSTRUCTION structure allows the repetition to be optimized and specifies that the operation will be followed by WCOUNT operands.

14.4.9 Operations Used to Render a Simple Triangle

Let's set up an Execute Buffer to render a simple triangle. For a simple triangle we will need three vertices in the vertex list. We will also need at least three operations. Two of the operations—D3DOP_TRIANGLE and D3DOP_EXIT—are straightforward. The third, D3DOP_PROCESSVERTICES, is needed to tell the 3D driver that we will provide vertices that don't need transformation or lighting. Both the D3DOP_TRIANGLE and D3DOP_PROCESSVERTICES operations are followed by operand structures.

Figure 14-5 shows a picture of the Execute Buffer we need to render our simple triangle. Now let's create this Execute Buffer. Our code will build the instruction stream in system memory. We will then create an Execute Buffer and copy the system memory buffer into the Execute Buffer using the code that we showed in 14.4.5.

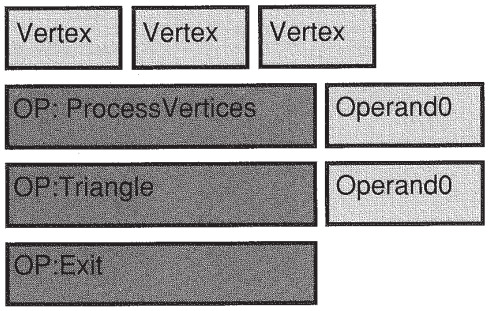


FIGURE 14-5 An Execute Buffer to render our simple triangle.

Here's the code that computes the size of the Execute Buffer needed and then allocates a system memory, or sysmem, buffer of this size:

```
// Create an execute buffer in system memory to render 1 triangle
size_t sztEx = 0;
sztEx = sizeof(D3DTLVERTEX) * 3;    // 3 vertices for a triangle
sztEx += sizeof(D3DINSTRUCTION)*3;  // processVerts, tri, exit
sztEx += sizeof(D3DPROCESSVERTICES); // 1 processVerts operand
sztEx += sizeof(D3DTRIANGLE) * 1;   // 1 triangle operand
m_pSysExBuffer = new BYTE [sztEx];  // setup exec buffer in sysmem first
```

Let's insert three vertices into our sysmem buffer.

We've picked a triangle of an arbitrary size and decided to color it green. In addition, to keep our example simple, we will instruct the driver not to transform or light the vertices. Our code provides pre-lit vertices in screen coordinates using a `D3DTLVERTEX` structure (instead of a standard `D3DVERTEX` structure):

```
// get ready to insert vertices
D3DTLVERTEX *pVerts = (D3DTLVERTEX *)m_pSysExBuffer;

    // V 0
    pVerts[0].dvSX = D3DVAL(10.0)
    pVerts[0].dvSY = D3DVAL(10.0);
    pVerts[0].dvSZ = D3DVAL(0.1);
    pVerts[0].dvRHW = D3DVAL(1.0);
    pVerts[0].dcColor = RGBA_MAKE(128, 255, 128, 0);
pVerts[0].dcSpecular = 0;
pVerts[0].dvTU = 0;
pVerts[0].dvTV = 0;

// V 1
    pVerts[1].dvSX = D3DVAL(300.0);
    pVerts[1].dvSY = D3DVAL(50.0);
    pVerts[1].dvSZ = D3DVAL(0.1);
    pVerts[1].dvRHW = D3DVAL(1.0);
    pVerts[1].dcColor = RGBA_MAKE(128, 255, 128, 0);
pVerts[1].dcSpecular = 0;
pVerts[1].dvTU = 0;
pVerts[1].dvTV = 0;

    // V 2
    pVerts[2].dvSX = D3DVAL(150.0);
    pVerts[2].dvSY = D3DVAL(180.0);
    pVerts[2].dvSZ = D3DVAL(0.1);
    pVerts[2].dvRHW = D3DVAL(1.0);
    pVerts[2].dcColor = RGBA_MAKE(128, 255, 128, 0);
pVerts[2].dcSpecular = 0;
pVerts[2].dvTU = 0;
pVerts[2].dvTV = 0;
```

In a D3DTLVERTEX structure, dvSX, dvSY and dvSZ specify screen coordinates of the vertex. Note that the fields are in floating point.

dvRHW is the reciprocal of homogeneous w. You can compute this value as 1 divided by the distance from origin to vertex along the *z*-axis.

dcColor sets the color of a vertex. With flat shading, all pixels in a triangle are set to the first vertex color, and the other two colors are ignored. With Gouraud shading, pixel colors are interpolated from the three vertex colors.

dcSpecular sets the reflectiveness of the material. You can use this field to add a metallic look to objects. We will experiment with it in Chapter 15.

dvTU and dvTV are used to map a vertex into texture coordinates. Here again, we will use these fields in Chapter 15.

Now let's enter our three operations after the vertices. The Direct3D SDK has a helper file (*d3dmacs.h*) with macros for inserting operations into an Execute Buffer. These macros do a decent job, and we have used them in our examples. We recommend that you take some time to look at these Direct3D macros.

```
pInsStart = m_pSysExBuffer + 3*sizeof(D3DTLVERTEX);  ◊ Remember where instructions start.
pTmp = (PVOID pIns);                                 ◊ Convert to void pointer for d3d macros.
```

The OP_XXX macros below take both a count and a void pointer as parameters. Once used, the macros increment the void pointer to point to the next valid location. Count is used to set the wCount field of the D3DINSTRUCTION as explained earlier. Remember to follow each opcode with the correct number of operands. Also note that macros for more complex opcodes require other parameters in addition.

```
// make sure vertices are copied to device memory
OP_PROCESS_VERTICES(1, pTmp);
    PROCESSVERTICES_DATA(D3DPROCESSVERTICES_COPY, 0, 3, pTmp);
```

The first parameter passed to OP_PROCESS_VERTICES indicates how the vertices should be processed. Four important options are

D3DPROCESSVERTICES_COPY	Vertices should simply be copied—they have been transformed and lit.
D3DPROCESSVERTICES_TRANSFORM	Vertices should be transformed.
D3DPROCESSVERTICES_TRANSFORMLIGHT	Vertices should be transformed and lit.
D3DPROCESSVERTICES_NOCOLOR	Vertices should not be colored.

ProcessVertices needs to know where to start and the number of vertices to process. The start vertex is specified by its index position.

```
// render triangle
OP_TRIANGLE_LIST(1, pTmp);
    ((LPD3DTRIANGLE)pTmp)->v1 = 0;
    ((LPD3DTRIANGLE)pTmp)->v2 = 1;
    ((LPD3DTRIANGLE)pTmp)->v3 = 2;
    ((LPD3DTRIANGLE)pTmp)->wFlags = 0;
pTmp = ((char*)pTmp) + sizeof(D3DTRIANGLE);
```

Vertices of a triangle are specified as a Word-sized index into the vertex list.

In addition to the vertices, triangle opcodes take a wFlags parameter that can be used to control how edges are drawn in wire-frame, strip, and fan modes. This is an advanced topic and is left as an extra credit exercise.

```
// exit operation
OP_EXIT(pTmp);          ◊ Terminate Execute Buffer list with OPEXIT.
pInsEnd = (char*)pTmp;  ◊ Remember where instructions end.
```

The following code is the same as what we listed in 14.4.5 to create an Execute Buffer and copy over an instruction from a system memory buffer. We've copied it here for your convenience:

```
// Describe ExecuteBuffer to be created
D3DEXECUTEBUFFERDESC debDesc;
memset(&debDesc, 0, sizeof(debDesc));
debDesc.dwSize = sizeof(debDesc);
debDesc.dwFlags = D3DDEB_BUFSIZE;
debDesc.dwBufferSize = sztEx;

// Create ExecuteBuffer and copy system buffer over
m_p3dFns->CreateExecuteBuffer(&debDesc, &m_pExBufFns, NULL);
m_pExBufFns->Lock(&m_ExDesc);              // lock buffer
pTmp = m_ExDesc.lpData;                    // get returned ptr
memcpy(pTmp, m_pSysExBuffer, sztEx);       // copy data over
m_pExBufFns->Unlock();                     // unlock buffer
```

```
// describe make-up of recently copied execute buffer
D3DEXECUTEDATA ExecData;
memset(&ExecData, 0, sizeof(D3DEXECUTEDATA));
ExecData.dwSize = sizeof(D3DEXECUTEDATA);
ExecData.dwVertexOffset = 0;
ExecData.dwVertexCount = 3;
ExecData.dwInstructionOffset = pInsStart - m_pSysExBuffer;
ExecData.dwInstructionLength = pInsEnd - pInsStart;
m_pExBufFns->SetExecuteData(&ExecData);
```

Whew! We're done setting up. Now we are ready to run.

14.4.10 Executing the Execute Buffers

3D graphics accelerators are often integrally linked to the system's standard graphics card. Hardware resources can be shared between 2D and 3D drivers. The device drivers may like to "context-swap," if you will, between the 2D and 3D tasks. To enable this context-swapping, Direct3D requires that the *IDirect3DDevice::Execute()* function be bracketed by *IDirect3DDevice::BeginScene()* and *IDirect3DDevice::EndScene()* calls.

Let's execute!

```
// "execute" the execute buffer
m_p3dFns->BeginScene();
m_p3dFns->Execute(pExecCmds, m_p3dViewport, D3DEXECUTE_UNCLIPPED);
m_p3dFns->EndScene();
```

IDirect3DDevice::Execute() takes three parameters: an IDirect3DExecuteBuffer object, an IDirect3DExecuteBuffer object, and a DWORD with modifier flags. With the *Execute* function being defined as a member of the IDirect3DDevice object, you can create multiple Execute Buffers to represent multiple objects and render objects selectively within a scene.

Similarly, since the viewport must be specified with each *Execute()* call, you could use a single Execute Buffer with different viewports to get different views of the object; alternatively, you could also render objects with differing viewing positions within a single scene.

The only modifier flags supported on *Execute()* are D3DEXECUTE_CLIPPED or D3DEXECUTE_UNCLIPPED. If you know that all objects in the Execute Buffer will fit within the 2D screen coordinates, you can improve performance by setting the flags to D3DEXECUTE_UNCLIPPED.

Be careful when executing Execute Buffers: They are hard to debug. If your application crashes while executing a set of instructions, all you know is that there was an error. The best that you can hope for is that the error didn't lock your machine into an unrecoverable state. Save often! Set yourself up for trial-and-error debugging. Start with small Execute Buffers and increase the size and complexity in small steps.

14.4.11 Seeing Results from 3D Devices

At this point, our Execute Buffer should render a triangle—we debugged it, so we know it is error-free. The 3D device will render the triangle into the 2D surface that it was "extended from." To actually see the triangle, we've got to make the 2D surface visible. This step requires standard DirectDraw programming that we learned in Part II. Here's the code to do it:

```
BOOL CSurface3d::UpdateScreen(LPDIRECTDRAWSURFACE2 pPrimary)
{
    // offset dst rect for client area position on primary surface
    long lRight = m_ptZeroZero.x + m_dwWidth;
    long lBottom = m_ptZeroZero.y + m_dwHeight;
    RECT rDst = {m_ptZeroZero.x, m_ptZeroZero.y, lRight, lBottom};
    RECT rSrc = {0, 0, m_dwWidth, m_dwHeight};

    // Blt with WAIT-UNTIL-BLITTER IS READY and no effects
    pPrimary->Blt(&rDst, m_p2dFns, &rSrc, DDBLT_WAIT, NULL);
    return TRUE;
}
```

Oh! Just one more thing, our display is in an RGB palette mode, and we'll need to realize the colors used to render our 3D object. If you remember the code way back in Section 14.4.3—we were forced to create a palette and attach it to the 2D surface before Direct3D would allow us to successfully create a 3D device. We even arranged the palette to permit the 3D device to modify palette colors. But this palette is attached to our Offscreen surface and does not automatically get realized.

Here's the code that realizes the palette by invoking *SetPalette()* on the Primary surface:

```
void CView::OnActivateView(…)
{
    // reset palette on primary surface
    LPDIRECTDRAWSURFACE2 pPrimary = gpAppWide->m_pGrfxCard->m_pSurfFns;
    LPDIRECTDRAWPALETTE  pPalette = gpAppWide->m_pGrfxCard->m_pPalette;
    if (pPrimary && pPalette) pPrimary->SetPalette(pPalette);
}
```

14.5 Demo Time

Try running the demo for this chapter on the CD. You should see a triangle appear on the screen. Move the mouse around and the triangle will follow the mouse. You have now worked through enough code to have an idea of how to get a triangle rendered using Direct3D's Immediate mode. Congratulations!

WHAT HAVE YOU LEARNED?

We spent some time filling you in on the background of 3D on the PC, primarily to demonstrate that the field is still in its infancy: The evolution has begun, the pace will be furious, the best is yet to come, and Direct3D is the foundation for the evolution. Within this foundation, we saw the two 3D modes that Direct3D offers: Retained mode for high-level abstraction, and Immediate mode for high performance.

If you worked through the code samples, you have

- handled code to get access to Direct3D functionality and 3D devices,
- linked Direct3D devices to DirectDraw surfaces and DirectDraw,
- set up a simple viewport to map a 3D world to 2D screen coordinates, and
- finally, you have created an Execute Buffer to render a triangle with a 3D device to a 2D surface (using the viewport and palette).

And now you're prepared . . . prepared for the next chapter on how to extend our simple triangle with texture mapping.

CHAPTER 15

Embellishing Our Triangle with Backgrounds, Shading, and Textures

WHY READ THIS CHAPTER?

In the previous chapter, we walked you through the bare minimum code needed to render a triangle with Direct3D. Our triangle from that chapter was solid filled, with a flat shaded color. What's more, we didn't redraw the background, and moving the triangle around left "triangle trails." Let's add some bells and whistles to our simple triangle.

In this chapter, you will

- learn how to use Direct3D to repaint the background and get rid of the "triangle trails" (we could have used a 2D background from Part II—but you'll need to understand how to use 3D backgrounds when you add lighting to your 3D scenes);
- play around with shading options and vary the coloring of triangles;
- load and use a texture to render the triangle with texture mapping; and
- understand the benefits of Z-Buffering and learn how to use Z-Buffering while rendering triangles.

PART V

15.1 Continuing Our Look into Direct3D

Over the course of this chapter, we will come across the Direct3D objects listed below in Table 15-1. We will describe the objects and the structures they use as we get to them.

Structures in Direct3D often contain unions of two naming conventions. Let's take, for example, the `D3DTRIANGLE` structure that we used as an operand in the last chapter. (See code on the next page.)

TABLE 15-1 Direct3D Objects Used in This Chapter

Object	Description
IDirect3DMaterial	Coloring options, such as color and texture
IDirect3DTexture	DirectDraw surface containing a texture map image

```
typedef struct _D3DTRIANGLE {
    union {
        WORD v1;
        WORD wV1;
    };
    union {
        WORD v2;
        WORD wV2;
    };
    union {
        WORD v3;
        WORD wV3;
    };
    WORD     wFlags;
} D3DTRIANGLE
```

Notice that each vertex in the structure is a union of two Word fields with different names for both Hungarian and non-Hungarian naming conventions. To simplify our discussions of structures, we will only show the Hungarian version and drop the unions.

The simplified `D3DTRIANGLE` structure would therefore be:

```
typedef struct _D3DTRIANGLE {
    WORD wV1;
    WORD wV2;
    WORD wV3;
    WORD wFlags;
} D3DTRIANGLE
```

15.2 Repainting the Background Using Direct3D

Let's continue where we left off in the previous chapter: Backgrounds weren't redrawn, and moving our triangle around left "triangle trails." Let's use Direct3D to repaint the background and get rid of the "triangle trails."

Why use Direct3D and why not use a 2D background from Part II? At some later stage you may want to add a spotlight into your 3D model. Moving the spotlight around might cause it to shine past 3D objects and onto some backdrop. Direct3D needs to know about this background to be able to illuminate it correctly. We'll use Direct3D to redraw this background.

Unlike triangles, backgrounds in Direct3D are not individual objects. Instead they are controlled as a method of an IDirect3DViewport object. The *IDirect3DViewport::SetBackground()* method takes a *material handle* as a parameter. So let's learn about materials.

15.2.1 Looking at Direct3D Materials

Even though lighting is computed by the Lighting module, some rendering methods are influenced by lighting factors. Lighting options in the Rendering module are controlled by the `D3DOP_STATELIGHT` opcode. Light state options that can be changed with this opcode are Fog, Ambient, and Material (defined as `D3DLIGHTSTATETYPE`).

In the Rendering module, Fog and Ambient controls apply globally to all objects. The material control, on the other hand, controls lighting properties of specific objects. Lighting controls are specified using a `D3DMATERIAL` structure. Let's look at the simplified version of this structure:

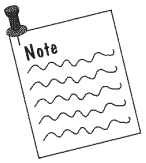
```
typedef struct _D3DMATERIAL {
    DWORD             dwSize;
    D3DCOLORVALUE     dcvDiffuse;
    D3DCOLORVALUE     dcvAmbient ;
    D3DCOLORVALUE     dcvSpecular;
    D3DCOLORVALUE     dcvEmissive;
    D3DVALUE          dvPower ;       ◊ Specify sharpness of specular reflections.
    D3DTEXTUREHANDLE  hTexture;       ◊ Combine a texture with specified coloring.
    DWORD             dwRampSize;     ◊ Shading gradient of colors in Ramp/Mono model.
} D3DMATERIAL, *LPD3DMATERIAL;
```

Four different color components.

The `D3DMATERIAL` structure provides dcvDiffuse, dcvAmbient, dcvEmissive, and dcvSpecular/dvPower to control four different color components of rendered objects. The `D3DTLVERTEX` structure also has dcColor and dcSpecular fields to control colors of rendered objects. The dual controls are combined during rendering. We will look at controlling colors later in this chapter.

Other fields in the `D3DMATERIAL` structure are: hTexture, through which we apply texture maps to a rendered object (discussed later in this chapter);

and dwRampSize, used by the Ramp/Mono driver to control fineness of shading in its approximated color model (to be seen in Chapter 16).



The D3DMATERIAL structure uses D3DCOLORVALUE type to define colors as opposed to the D3DCOLOR type used by D3DTLVERTEX structures. RGB values in D3DCOLORVALUE are floating point fields within a structure (normally ranging from 0.0 to 1.0); RGB values in D3DCOLOR are byte-sized values packed into a DWORD (ranging from 0 to 255).

15.2.2 Creating a Direct3D Background

Again, since backgrounds in Direct3D are not individually rendered objects, we cannot set the color of backgrounds with D3DTLVERTEX.dcColor (as we did for triangles). Instead, we set the color of backgrounds with the color fields of a D3DMATERIAL structure. (Given that materials can also contain a texture, we can render an image as a background using textures—but we're getting ahead of ourselves! We will get to texture mapping shortly.)

The field of interest presently is D3DMATERIAL.dcvDiffuse. It is diffuse reflections of light that give objects their basic color; such as a blue ball or a red box. With Direct3D materials we use *dcvDiffuse* to set the basic color of the material.

Let's use Direct3D to implement a CBackground3d class with a single color. We start by first creating and setting up a material and then associating the material with our viewport.

In Direct3D, materials are managed through an *IDirect3DMaterial* interface object. Here is the code to create an IDirect3DMaterial object and to set its properties with a D3DMATERIAL structure and the *IDirect3DMaterial::SetMaterial()* method:

```
BOOL CBackground3d::Init(LPDIRECT3D pD3D)
{
    // create a material for the background
    pD3D->CreateMaterial(&m_pMaterialFns, NULL);

    // init material descriptor
    memset(&m_MaterialDesc, 0, sizeof(m_MaterialDesc));
    m_MaterialDesc.dwSize = sizeof(m_MaterialDesc);

    // set diffuse coloring ("pinkish color")
    m_MaterialDesc.dcvDiffuse.dvR = D3DVALUE(0.85);
    m_MaterialDesc.dcvDiffuse.dvG = D3DVALUE(0.15) ;
    m_MaterialDesc.dcvDiffuse.dvB = D3DVALUE(0.50);
    m_MaterialDesc.dcvDiffuse.dvA = D3DVALUE(0.0);
```

IDirect3D::CreateMaterial() returns an IDirect3DMaterial interface object.

Set color of background by setting the dcvDiffuse component of material.

```
    // set material with above description
    m_pMaterialFns->SetMaterial(&m_MaterialDesc);
    return TRUE;
}
```

IDirect3DMaterial::SetMaterial() actually sets material properties.

And now we will set up our background with this material:

```
BOOL CBackground3d::Attach(LPDIRECT3DDEVICE p3dSurfFns, LPDIRECT3DVIEWPORT pView)
{
    // need material handle
    m_pMaterialFns->GetHandle(p3dSurfFns, &m_hMaterial);

    // set background of viewport
    pView->SetBackground(m_hMaterial);
    return TRUE;
}
```

For hardware acceleration devices, source data might have to reside in video memory. Video memory is limited, and Direct3D occasionally uses "handles" for memory allocation. The *GetHandle()* method loads an object onto the specified device and returns a handle to the loaded object.

15.2.3 Blitting a Direct3D Background

We have set up our viewport to have a background. Viewport backgrounds are not automatically drawn. The *IDirect3DViewport::Clear()* method is used to control the drawing of backgrounds. By using *Clear()*, we can control when the background gets redrawn. In addition, *Clear()* uses an array of rectangles to specify how much of the background gets redrawn, helping us reduce the cost of repainting backgrounds with appropriate "dirty rectangle" logic.

Here's our code to Blt backgrounds. We have not implemented any "dirty rectangle" logic. But in keeping with the CBackground class from Part II, we can specify a sub-rectangle to the Blt method to implement moving backgrounds.

```
BOOL CBackground3d::Blt(LPDIRECT3DVIEWPORT pView, POINT *pptDst, RECT *prSrc)
{
    // draw specified sub-rect of background at specified pt
    DWORD dwWidth  = prSrc->right - prSrc->left;
    DWORD dwHeight = prSrc->bottom - prSrc->top;

    //create a d3dRect to clear background (needs to be clipped)
    D3DRECT    drDst;
    drDst.x1 = pptDst->x;
    drDst.y1 = pptDst->y;
```

IDirect3DViewport::Clear() expects an array of D3DRECTs.

```
    drDst.x2 = drDst.x1 + dwWidth ;
    drDst.y2 = drDst.y1 + dwHeight;
    #define nRECTS 1

    // clear viewport to "draw" background
    pView->Clear(nRECTS, &drDst, D3DCLEAR_TARGET);
    return TRUE;
}
```

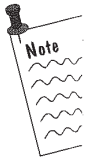
The first two parameters specify an array of rectangles to control "how much" gets cleared by *IDirect3DViewport::Clear()*. The third parameter to *IDirect3DViewport::Clear()* specifies flags to control "what" gets cleared—options are the rendering target or the Z-Buffer or both.

We are now all set to have the 3DDevice clear our background. As we mentioned in the previous chapter, 3DDevices may batch rendering operations and the actual clear (redraw) is only guaranteed to have been completed on return from the *IDirect3DDevice::EndScene()* call.

Try running the demo for this chapter on the CD. Voilà! No more triangle trails!

15.3 Controlling Shading Options

In the last chapter, we worked through the bare minimum code needed to render a triangle. We pretty much left the Rendering module in its default state. The Immediate mode Rendering module in Direct3D offers a lot of rendering options. Let's take a look at how to control these options and then play around with some of them.



15.3.1 Looking at Some Render States and Their Default Values

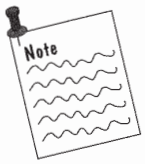
With Direct3D Immediate mode's `D3DOP_STATERENDER` opcode we can control various states of the Rendering module. Some of the straightforward Render states are listed in Table 15-2.

TABLE 15-2 Direct3D Render State Types

D3DRENDERSTATETYPE	Description
D3DRENDERSTATE_FILLMODE	Fill triangle, draw edges, draw vertices
D3DRENDERSTATE_SHADEMODE	Flat, Gouraud, (future: Phong)
D3DRENDERSTATE_TEXTUREHANDLE	Set mode to texture mapping and specify texture
D3DRENDERSTATE_DITHERENABLE	Enable/disable dithered coloring
D3DRENDERSTATE_SPECULARENABLE	Enable/disable specular highlights
D3DRENDERSTATE_ZENABLE	Enable/disable Z-Buffering

Now let's look at the default state of these render states and examine their impact on our simple triangle render example:

_FILLMODE	The default value is D3DFILL_SOLID, which is what we want for our simple triangle.
_SHADEMODE	The default is D3DSHADE_GOURAUD. The implication is that the colors of the vertices would have been interpolated across the intervening space. But if you look at the code, you'll see that we set all the vertices to the same color and got a flat-shaded effect.
_TEXTUREHANDLE	The default is NUL—which is how we turn off texture mapping and render shaded triangles. Note that there is no equivalent SHADEMODE setting to turn off shading.
_DITHERENABLE	The default is FALSE. If we'd opted for differing colors at the triangle vertices, we might have ended up with a banded picture on our low-resolution 8 bpp screen. Dithering improves the picture quality of Gouraud interpolation but reduces rendering performance.
_SPECULARENABLE	The default is TRUE. But we deliberately disabled specular highlights by setting dcSpecular of all vertices to 0.
_ZENABLE	The default is FALSE, which means that triangles will be rendered sequentially. Z-Buffering does not really affect our single triangle example. But if we rendered multiple triangles without Z-Buffering, we would have to order the triangles from back to front.



There are too many states to list here (three Transform states; seven Lighting states; and about seventy-two Render states including thirty-two stipple patterns). We strongly recommend that you look through the Direct3D documentation and familiarize yourself with the various states and their *default values*.

15.3.2 Coloring a Pixel in Direct3D

Direct3D drivers combine vertex components (such as color, specularity, and alpha blending) and render effects (such as fog, dither, anti-alias) to compute the final value of a pixel. Vertex components are specified at each vertex, and values at intervening pixels are computed from vertex components based on a shade mode. *Shade modes* currently permitted are flat and Gouraud. No interpolation is done with flat shading, and the component values at the first vertex of a triangle are applied across the entire triangle. With Gouraud shading, component values from the three triangle vertices are linearly interpolated to get the values at intervening pixels. Phong

shading, where all lighting is reevaluated for each pixel of a triangle, is not currently supported, but it has been defined for possible future support.

Based on the shade mode, an intervening pixel's color is computed from the RGB color values specified at the triangle's vertices. The meaning of the color values and the result of the interpolation varies with the color model used (RGB or Ramp). We'll examine this factor in more detail shortly.

A shaded pixel is further modified by adding its *specular* component. Specularity is specified as an RGB color value at each vertex. The specularity of intervening pixels is computed using the shade mode in effect. Once again, the approach to computing specularity and its result varies with the color model used. We'll look at this in more detail too.

When alpha blending is enabled the *alpha* component of a color is also interpolated according to the shade mode. However, pixel values are not affected by alpha interpolation if blending is implemented by stippling. Alternatively, pixel values are effected by alpha interpolation, if alpha blending is implemented by texture blending. (Note: When alpha components are not supported in a given mode, the alpha value of colors is implicitly 255. This is the maximum possible alpha; that is, alpha is at full intensity.)

When *texture mapping* is enabled, the source "texel" value also contributes to the pixel's value. As we mentioned earlier, shading cannot be turned off; therefore, the texel value is only a partial contributor. This contribution is "blended" with the value of the color and the specular components. Blending is controlled by `D3DRENDERSTATE_TEXTUREMAPBLEND`. The default value is `D3DTBLEND_MODULATE`, where RGB values of the texture are multiplied with the computed RGB values. (Alpha values in the texture supersede computed alpha values.)

One of the values that we can set `D3DRENDERSTATE_TEXTUREMAPBLEND` to is `D3DTBLEND_COPY`. In Copy mode, the renderer ignores color computations and simply copies texels to the screen; therefore, textures must have the same pixel format and the same palette as the primary surface. Copy mode effectively turns off shading and typically offers a significant performance boost. This is often a good technique to attain higher performance with pre-lit textured scenes.

15.3.3 Shading with the RGB Color Model

Direct3D offers two different color models: RGB and Ramp (Mono). The two models treat pixel coloring differently, offering varying quality versus performance options. Let's take a quick look at how shading is treated by the RGB color models.

The *RGB model* operates in true color space using 24 bits to combine red, green, and blue light. With Gouraud shading in the RGB color model, each of the red, blue, and green components is individually interpolated and then recombined to produce the shaded pixel. Specular values are similarly computed. The alpha component is independently interpolated to allow the driver to choose the interpolation technique that matches the implemented alpha blending approach.

We can use the 24-bit RGB driver to render to 8-, 16-, 24-, and 32-bit displays. However, banding artifacts are sometimes apparent when the RGB driver has to render a scene down to less than 24 bits. Turning on dithered rendering helps us reduce the apparent effects of banding artifacts.

15.3.4 Shading with the Ramp Color Model

Our code samples from both the previous chapter and the current one use the RGB color model. Programming with the Ramp model requires a little more explanation, and we will look at it in Chapter 16. But while we're on the subject of pixel coloring and shading models, why don't we take a quick look at how pixels are colored by the Ramp model.

The *Ramp model* operates through lookup tables. Colors have no real meaning. In fact, the Ramp driver uses only the blue component of an RGB color specification. When interpolating between two vertices, the Ramp driver interpolates this blue component with no regard to the actual color. The driver then accesses "a lookup table" to interpret the final result.

The Ramp driver builds lookup tables from material definitions. For materials with no specularity, the driver builds a "color ramp" ranging from the ambient color to the maximum diffuse color. For materials with specularity, the driver builds a two-stage color ramp; the first stage ranges from the ambient color to the maximum diffuse color, and the second stage ranges from maximum diffuse color light to the maximum specular color. For materials with textures, the Ramp driver builds a color ramp for each color in the texture.

We can control the size of the ramp with the dwRampSize field in the material definition. In a palletized display mode, each ramp entry equates to a palette entry. Increasing ramp sizes will increase shading resolution, but this method will consume valuable palette space. After all free palette entries are used up, the Direct3D system will find colors that most closely match the intended colors. Huge ramps or a large variety of colors can also cause poor caching and therefore degrade rendering performance.

15.3.5 Changing Default Render States

Let's play around with shading and changing render states. First, we will revisit our simple triangle and change vertex colors to see the effect of Gouraud shading.

```
// modify colors of vertices
D3DTLVERTEX *pVerts = (D3DTLVERTEX *)m_pSysExBuffer;
pVerts[0].dcColor = RGBA_MAKE(128, 255, 128, 0);
pVerts[1].dcColor = RGBA_MAKE(128, 0, 128, 0);
pVerts[2].dcColor = RGBA_MAKE(0, 255, 128, 0);
```

dcColor sets the color of each vertex. With Gouraud shading, pixel colors are interpolated from the three vertex colors. With flat shading, all pixels in a triangle are set to the first vertex color, and the other two colors are ignored.

Now run the demo for this chapter and check the Gouraud shading option. You should see the colors varying throughout our triangle. Do you see what we mean by banding artifacts?

Next we enable dithering, that is, set `D3DRENDERSTATE_DITHERENABLE` to `TRUE`. We change render states by using the `D3DOP_STATERENDER` operation. A single `D3DOP_STATERENDER` instruction controls one state variable and is followed by the state to be changed and its new value. (Use `D3DOP_STATETRANSFORM` to change transform states and `D3DOP_STATELIGHT` to change lighting states.)

15.4

Here's code that adds the `D3DOP_STATERENDER` instruction into our previous execute buffer:

Compute size of Execute Buffer needed for triangle and state change.

```
sztEx = sizeof(D3DTLVERTEX) * 3;      // 3 vertices for a triangle
sztEx += sizeof(D3DINSTRUCTION) * 4; // state, processVerts, tri, exit
sztEx += sizeof(D3DSTATE) * 1;        // 1 texture render state
sztEx += sizeof(D3DPROCESSVERTICES); // 1 process vertice operand
sztEx += sizeof(D3DTRIANGLE) * 1;     // 1 triangle operand
m_pSysExBuffer = new BYTE [sztEx];    // setup ex buffer in sysmem first
memset(m_pSysExBuffer, 0, sztEx);     // zero out sys mem buffer
```

Modify operations from last chapter to add render state operation.

```
LPBYTE lpInsStart = m_pSysExBuffer + sizeof(D3DTLVERTEX)*3;
LPVOID lpTmp = (LPVOID)lpInsStart;
OP_STATE_RENDER(1, lpTmp);
    STATE_DATA(D3DRENDERSTATE_DITHERENABLE, TRUE, lpTmp);
OP_PROCESS_VERTICES(1, lpTmp);
    PROCESSVERTICES_DATA(D3DPROCESSVERTICES_COPY, 0, 3, lpTmp);
OP_TRIANGLE_LIST(1, lpTmp);
    ((LPD3DTRIANGLE)lpTmp)->v1 = 0;
    ((LPD3DTRIANGLE)lpTmp)->v2 = 1;
    ((LPD3DTRIANGLE)lpTmp)->v3 = 2;
    ((LPD3DTRIANGLE)lpTmp)->wFlags = 0;
    lpTmp = ((char*)lpTmp) + sizeof(D3DTRIANGLE);
OP_EXIT(lpTmp);
LPBYTE lpInsEnd = (LPBYTE)lpTmp;
```

Insert D3DOP_STATERENDER using the OP_STATE_RENDER macro from *d3dmacs.h*

Run the demo for this chapter and toggle the Dither option. How's the quality? We'll measure performance in Chapter 16.

Well, now you know how to change render states. Why don't you try some out for yourself?

Try setting D3DRENDERSTATE_SHADEMODE to D3DSHADE_FLAT and see how only the first vertex color is used. Other options we're sure you can handle are D3DRENDERSTATE_ANTIALIAS and D3DRENDERSTATE_FILLMODE. Have at it!

15.4 Texture Mapping with Direct3D

OK! How about we whip up a batch of some texture mapping?

15.4.1 Creating a Texture Map

To start out, texture maps, like IDirect3DDevices, are "converted" DirectDraw surfaces. So to create an IDirect3DTexture object, we first create an IDirectDrawSurface and then use *QueryInterface* to retrieve an IID_IDirect3DTexture interface.

Pixel format allowed for texture surfaces varies based on the Direct3D driver being used. With the RGB HEL driver, we can create textures of various bit depths (including 1-, 2-, 4-, 8-, 16-, 24-, and 32-bit textures) and various formats (such as DDPF_RGB, DDPF_PALETTEINDEXED8). Check the Direct3D documentation for a complete list of formats supported for texture maps.

With the Mono/Ramp HEL driver, textures must either be palletized textures or be textures of the same format as those of the primary display surface. Hardware acceleration devices can also support a variety of formats for texture maps. Surfaces must be enumerated to find out the supported formats.

We will work with the Ramp HEL driver in Section 18.3. For now let's continue using the RGB HEL driver with 8 bpp surfaces. Here's code that demonstrates how to create a surface suitable for texture mapping. The code then "converts" the surface to create an IDirect3DTexture object, and finally it loads a bitmap image into the texture.

```
BOOL CTriangleTex::Init(LPDIRECTDRAW2 pDDraw, LPDIRECTDRAWPALETTE pPalette, UINT nRes)
{
```

Standard GDI code to load bitmap data from resource.

```
CBitmap    cBmp;
cBmp.LoadBitmap(nRes);
BITMAP bm;
cBmp.GetBitmap(&bm);
pData = new BYTE[bm.bmHeight * bm.bmWidth];
cBmp.GetBitmapBits(bm.bmWidth * bm.bmHeight, pData);
```

Setup to create a structure suitable for texture maps.

```
m_SurfDesc.dwHeight = roundUpPowerOfTwo(bm.bmHeight);
m_SurfDesc.dwWidth =  roundUpPowerOfTwo(bm.bmWidth);
m_SurfDesc.ddpfPixelFormat.dwRGBBitCount = 8;
m_SurfDesc.ddpfPixelFormat.dwFlags = DDPF_PALETTEINDEXED8 | DDPF_RGB;
m_SurfDesc.ddpfPixelFormat.dwSize = sizeof(DDPIXELFORMAT);
m_SurfDesc.ddsCaps.dwCaps = DDSCAPS_TEXTURE;
m_SurfDesc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT;
m_SurfDesc.dwFlags |= DDSD_CAPS | DDSD_PIXELFORMAT;
```

Specify that this surface will be used for texture mapping.

Standard DirectDraw code to create a surface and load an image into the surface's data space.

```
LPDIRECTDRAWSURFACE    pDD1Surf;
pDDraw->CreateSurface(&m_SurfDesc, &pDD1Surf, NULL);
pDD1Surf->Lock(NULL, &m_SurfDesc, DDLOCK_WAIT, NULL);
PBYTE pDst = (PBYTE)m_SurfDesc.lpSurface;
PBYTE pSrc = pData;
for (DWORD dwRow = 0; dwRow < m_dwHeight; dwRow++) {
    memset(pDst, 0, m_SurfDesc.lPitch);
    memcpy(pDst, pSrc, m_dwWidth);
    pDst += m_SurfDesc.lPitch;
    pSrc += bm.bmWidth;
}
pDD1Surf->Unlock(NULL);
pDD1Surf->SetPalette(pPalette);

// Convert to TextureObject and get its handle
pDD1Surf->QueryInterface(IID_IDirect3DTexture, &m_pSurfFns);
pDD1Surf->Release();
m_pSurfFns->GetHandle(p3dFns, &m_hTexture);

// free temporary memory, return
delete pData;
return TRUE;
}
```

Our texture map surface has a DDPF_PALETTEINDEXED8 pixel format. Direct3D requires us to set the palette before getting an IDirect3DTexture interface. Our application has been designed to use the same palette on all objects.

QueryInterface on the IDirectDrawSurface gives us an IDirect3DTexture interface to the same object. We now have two separate interfaces to the same object. Both interfaces must be released for the object to be completely freed. In this simple example, we will not use any DirectDraw functionality on the texture surface; therefore, we have used a local variable for temporary access to the IDirectDrawSurface, and we are releasing the interface before we exit. In later examples we will use DirectDraw functionality on the texture surface and then even the IDirectDrawSurface will be part of the CSpriteTex object.

The Height and Width of texture map surfaces used for texture mapping *must* be a power of two. *CreateSurface()* will successfully create a surface with non-"power-of-two" dimensions. But later at *Execute()* time, the rendering engine will crash, and there will be no indication of the nature of the error.

IDirect3DTexture::GetHandle() loads a texture into device memory and returns a handle to the object. To provide more control of device memory use, *IDirect3DTexture* has *Load()* and *Unload()* methods. These methods will work only with surfaces created with the DDSCAPS_ALLOCONLOAD flag set in the ddsCaps.dwCaps field of the surface descriptor.

EXTRA CREDIT

If you're looking for an extra credit opportunity, read the Direct3D documentation on texture compression and write code to compress a texture. Here's a hint: DDSCAPS_ALLOCONLOAD and *Load()*. Go on! You can do it!

15.4.2 Setting Up Triangle Vertices for Texture Mapping

So far we've created and loaded a texture. Now using code that we've mostly seen in the previous chapter, let's set up an Execute Buffer to render a texture-mapped triangle:

```
BOOL CTriangleTex::Init(LPDIRECTDRAW2 pDDraw, LPDIRECT3DDEVICE p3dFns, UINT nResID)
{
    Compute size of Execute Buffer needed to render simple texture-mapped triangle.
    sztEx = sizeof(D3DTLVERTEX) * 3;     // 3 vertices for a triangle
    sztEx += sizeof(D3DINSTRUCTION) * 4; // state, processVerts, tri, exit
    sztEx += sizeof(D3DSTATE) * 1;       // 1 texture render state
    sztEx += sizeof(D3DPROCESSVERTICES); // 1 process vertice operand
    sztEx += sizeof(D3DTRIANGLE) * 1;    // 1 triangle operand
    m_pSysExBuffer = new BYTE [sztEx];   // setup ex buffer in sysmem first
    memset(m_pSysExBuffer, 0, sztEx);    // zero out sys mem buffer
    Set up vertex info for texture-mapped triangle.
    D3DTLVERTEX *pVerts = (D3DTLVERTEX *)m_pSysExBuffer;
    // V 0
    pVerts[0].dvSX = D3DVAL(0.0);
    pVerts[0].dvSY = D3DVAL(0.0);
    pVerts[0].dvSZ = D3DVAL(0.1);
    pVerts[0].dvRHW = D3DVAL(1.0);
    pVerts[0].dcColor = RGBA_MAKE(255, 255, 255, 255) ;
                                  For now, set color value to white. We will
                                  explain this later in the chapter.
    pVerts[0].dvTU = D3DVAL(0.0);
    pVerts[0].dvTV = D3DVAL(0.0);
                                  Specify in floating point values how each triangle vertex maps to the texture. Texture
                                  itself ranges from 0.0 (top, left) to 1.0 (right, bottom).
    // V 1
    pVerts[1].dvSX = pVerts[0].sx + D3DVAL(300.0);
    pVerts[1].dvSY = pVerts[0].sy + D3DVAL(100.0);
    pVerts[1].dvSZ = D3DVAL(0.1);
    pVerts[1].dvRHW = D3DVAL(1.0);
    pVerts[1].dcColor = RGBA_MAKE(255, 255, 255, 255);
    pVerts[1].dvTU = D3DVAL(1.0);
    pVerts[1].dvTV = D3DVAL(1.0);
    // V 2
    pVerts[2].dvSXsx = pVerts[0].sx + D3DVAL(150.0);
    pVerts[2].dvSYsy = pVerts[0].sy + D3DVAL(180.0);
    pVerts[2].dvSZsz = D3DVAL(0.1);
    pVerts[2].dvRHrhw = D3DVAL(1.0);
    pVerts[2].dcColor = RGBA_MAKE(255, 255, 255, 255);
    pVerts[2].dvTU = D3DVAL(0.0);
    pVerts[2].dvTV = D3DVAL(1.0);
```

The difference in setting up the vertices is the setting of texture coordinates in the dvTU and dvTV fields of the `D3DTLVERTEX` structure. For each vertex, we need to specify how it maps to the texture. All textures, no matter their size, are defined to range from 0.0 to 1.0. Values for dvTU and dvTV need not lie within the "0.0 to 1.0" range. We can specify any legal floating point value, either negative or positive.

How the rendering engine reacts to texture addresses outside the "0.0 to 1.0" range depends on the D3DRENDERSTATE_TEXTUREADDRESS state variable. Valid values are D3DTADDRESS_WRAP, D3DTADDRESS_MIRROR, and D3DTADDRESS_CLAMP. The default state is D3DTADDRESS_WRAP. Refer to the Direct3D documentation for a deeper understanding of each state.

15.4.3 Setting Up Render Operations for Texture Mapping

Now we will set up the operations to render our texture-mapped triangle. The code for this procedure again is pretty much the same as that from the previous chapter. The significant difference is that we've now got to tell the rendering engine to use texture mapping while rendering, and we need to tell it which texture to use.

Looking through the various render states, we come across the D3DRENDERSTATE_TEXTUREHANDLE state type. The TEXTUREHANDLE state type with a D3DOP_STATERENDER opcode tells the rendering engine to use texture mapping, and the operand specifies which texture to use:

Modify operations from the last chapter to render texture-mapped triangle.

```
LPBYTE lpInsStart = m_pSysExBuffer + sizeof(D3DTLVERTEX)*3;
LPVOID lpTmp = (LPVOID)lpInsStart;
OP_STATE_RENDER(1, lpTmp);
    STATE_DATA(D3DRENDERSTATE_TEXTUREHANDLE, m_hTexture, lpTmp);
OP_PROCESS_VERTICES(1, lpTmp);
    PROCESSVERTICES_DATA(D3DPROCESSVERTICES_COPY, 0, 3, lpTmp);
OP_TRIANGLE_LIST(1, lpTmp);
    ((LPD3DTRIANGLE)lpTmp)->v1 = 0;
    ((LPD3DTRIANGLE)lpTmp)->v2 = 1;
    ((LPD3DTRIANGLE)lpTmp)->v3 = 2;
    ((LPD3DTRIANGLE)lpTmp)->wFlags = 0;
    lpTmp = ((char*)lpTmp) + sizeof(D3DTRIANGLE);
op_exit(lpTmp);
lpbyte lpInsEnd = (LPBYTE)lpTmp;
```

Tell the rendering engine to render all of the following triangles using texture mapping with the texture specified in the parameter. Once again, our code sets up Execute Buffer operations with the macros in *d3dmacs.h*.

Finally let's create our Execute Buffer, copy our instruction stream into device data space, and then describe our buffer to the 3D device. This code is identical to code we've seen before:

Same code as in the last chapter to set up an Execute Buffer on a 3D device.

```
m_ExDesc.dwFlags = D3DDEB_BUFSIZE;
m_ExDesc.dwBufferSize = m_sztEx;
p3dFns->CreateExecuteBuffer(&m_ExDesc, &m_pExBufFns, NULL);

// copy triangle into execute buffer space
m_pExBufFns->Lock(&m_ExDesc);
lpTmp = (LPBYTE)m_ExDesc.lpData;
memcpy(lpTmp, m_pSysExBuffer, lpInsEnd-m_pSysExBuffer);
m_pExBufFns->Unlock();

// describe execute buffer to 3ddevice
m_ExData.dwVertexOffset = 0;
m_ExData.dwVertexCount = 3;
m_ExData.dwInstructionOffset = lpInsStart - m_pSysExBuffer;
m_ExData.dwInstructionLength = lpInsEnd - lpInsStart;
m_pExBufFns->SetExecuteData(&m_ExData);

return TRUE;
}
```

Run the demo for this chapter and check the Texture Mapping option. You should see a texture-mapped triangle chasing the mouse around.

15.4.4 Handling "Lit" Texture Maps

We'd like to get back to something that we brought up earlier but left for later. Remember when we were setting up vertices for texture mapping (in 15.4.2), we set the dcColor field of all the vertices to `WHITE` and said that we would explain it later.

We mentioned earlier that

- the texel contribution is "blended" with the computed pixel value;
- blending is controlled by `D3DRENDERSTATE_TEXTUREMAPBLEND`; and
- the default blending state is `D3DTBLEND_MODULATE`.

`D3DTBLEND_MODULATE` *multiplies* source texels by computed values. Colors within the Rendering module are treated as values from 0 to 1. Multiplying values in these ranges will produce smaller results (unless either value is 1). Therefore modulation reduces the brightness of color components unless one of the source components is 1. The simplest way to ensure that the colors of a texture do not change during texture mapping is to set dcColor to `WHITE` (255, 255, 255) and to set dcSpecular to `BLACK` (0, 0, 0).

An alternate way to ensure that the colors of a texture do not change during texture mapping is to change the RENDERSTATE_TEXTUREMAPBLEND render state to D3DTBLEND_COPY. In Copy mode, the renderer ignores color computations and simply copies texels to the screen. But the textures must have the same pixel format as the Primary surface, and they also must also use the same palette. Our simple triangle application has been set up this way.

You should find changing render states to be pretty trivial by now. Here's the code that we insert into our Execute Buffer to use Copy mode:

```
// Don't forget to increase the size of execute buffer needed
// change render state to use Copy mode
OP_STATE_RENDER(1, lpTmp);
    STATE_DATA(D3DRENDERSTATE_TEXTUREMAPBLEND, D3DTBLEND_COPY, lpTmp);
```

Run the demo for this chapter and turn on the Copy mode option. Toggling Gouraud shading should have no effect. Turn Copy mode off then toggle Gouraud shading. Notice how the texture map colors have become duller?

One last point before we move on from texture mapping. Since a texture is also a DirectDraw surface, we can render into the texture using DirectDraw surface functions, such as *Lock()* and *Unlock()*, and then texture map this data onto a 3D object.

15.5 Z-Buffering with Direct3D

15.5.1 Why Bother with Z-Buffering?

Take a look at the two triangles in Figure 15-1. Let's assume we have an Execute Buffer that has triangle 1 inserted first and triangle 2 inserted second. Without Z-Buffering, the triangles are rendered in the order that they are encountered. Triangle 2 will be drawn after triangle 1, and therefore it will be drawn on top of triangle 1 as shown in the figure.

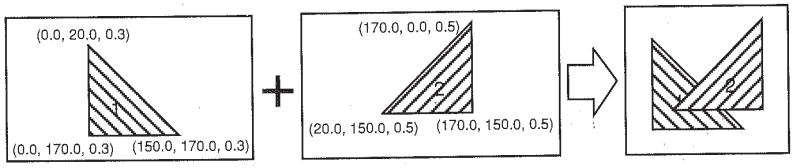


FIGURE 15-1 Triangles rendered from back to front regardless of Z-values.

Now let's take a look at the Z-values of the triangles. All the Z-values in triangle 1 are 0.3, and all the Z-values in triangle 2 are 0.5. But viewport coordinates are defined to go from 0.0 in front to 1.0 in the back, so by this definition triangle 1 should have been drawn in front of triangle 2. Instead, the renderer ignored the Z-values to render the triangles. (Without Z-Buffering, Z-values are only used to correct perspective while texture mapping.)

Without Z-Buffering, it is our responsibility to sort the triangles and insert them in the correct order. The compute expense to re-sort triangles may be very expensive for some application scenarios. (For instance, for 3D models with many overlapping objects and complete freedom of movement. In these scenarios it may be preferable to use Z-Buffering, so that the triangles will be rendered according to their Z-values and regardless of the sort order.)

Let's take a look at a second example as shown in Figure 15-2.

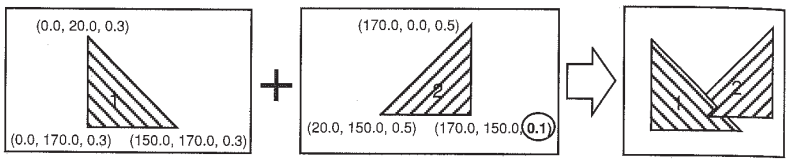


FIGURE 15-2 Intersecting triangles rendered with Z-Buffering.

The ellipse highlights the Z-value of the second vertex in triangle 2. We've changed this Z-value from the previous value of 0.5 to a new value of 0.1. This vertex is now *in front* of the vertices of triangle 1. As a result, triangle 2 now partially overlaps triangle 1.

Once again, without Z-Buffering the renderer ignores Z-values, and it will render the triangles without any overlap. *Without Z-Buffering, it is our responsibility to split up intersecting triangles.* With complex models, we may want to opt for Z-Buffering, because the renderer will test the Z-value of each pixel that it draws, and intersecting triangles will be correctly rendered.

15.5.2 Setting Up for Z-Buffering

Let's look at the code needed to set up and use Z-Buffering. Z-Buffers in Direct3D are merely another form of DirectDraw surfaces. Unlike with 3D devices and texture maps, there is no functionality applicable directly to Z-Buffers, so there is no need to create a new interface object.

So creating a Z-Buffer is as simple as creating a DirectDraw surface. The only point worth mentioning is that we don't get to choose the Z-Buffer pixel depth—this choice is made by the 3D driver. Well, how do we find out about the 3D device's choice? There are two ways: use *IDirect3DDevice::GetCaps()* or remember it from section 14.4.2, when we enumerated device drivers.

In the code for this chapter, we've inserted our Z-Buffer analysis into our device enumeration callback. Here's the snippet of code that was inserted:

Examine the chosen LPD3DDEVICEDESC structure for Z-Buffering support.

```
pGrfx->m_bCanZbuf = FALSE;
if (pChoice->dwDeviceZBufferBitDepth != 0) {
    pGrfx->m_bCanZbuf = TRUE;
    // bit depth is in DDBD format
    DWORD ddbd = pChoice->dwDeviceZBufferBitDepth;
    // convert to bpp format
    pGrfx->m_dwZBufferBPP = cvtToBPP(ddbd);
}
```

Test for Z-Buffering support by looking at the dwDeviceZBufferBitDepth field.

Even though dwDeviceZBufferBitDepth is specified using the packed DDBD format, the documentation for D3DDEVICEDESC states that this field can only be *one of* these formats: DDBD_8, DDBD_16, DDBD_24, or DDBD_32. Packed DDBD format are #defines that need to be converted to get pixel depth.

Now that we know that our device can Z-Buffer (and we know the depth of its Z-Buffer), we can go on to create a Z-Buffer using straightforward DirectDraw code:

```
BOOL CSurface3d::InitZbuffer(LPDIRECTDRAW2 pDDraw, DWORD dwZBufferBPP)
{
```

Setup descriptor for Z-Buffer (a special form of DirectDraw surface).

```
    //use same width and height as 3d surface; use specified bpp
    m_ZSurfDesc.dwHeight = m_dwHeight;
    m_ZSurfDesc.dwWidth = m_dwWidth;
    m_ZSurfDesc.dwZBufferBitDepth = dwZBufferBPP;
    // set caps flag
    m_ZSurfDesc.ddsCaps.dwCaps = DDSCAPS_ZBUFFER;
    DWORD dwMem = (m_bIsVidMem) ? DDSCAPS_VIDEOMEMORY : DDSCAPS_SYSTEMMEMORY;
    m_ZSurfDesc.ddsCaps.dwCaps |= dwMem;
    // set which fields in structure were valid
    m_ZSurfDesc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT;
    m_ZSurfDesc.dwFlags |= DDSD_CAPS | DDSD_ZBUFFERBITDEPTH;
```

Request Z-Buffer.

Put Z-Buffer in same memory as 3D surface.

```
    // Create Z-buffer using DirectDraw create surface
    pDDraw->CreateSurface(&m_ZSurfDesc, &m_pZSurfFns, NULL);
    return TRUE;
}
```

Create the Z-Buffer!

We've created our Z-Buffer, but as of yet it's floating around freely, single and unattached. We've got to marry it to our 3D surface. It so happens that we cannot attach the Z-Buffer directly to the 3D surface. Instead we've got to use a matchmaker—the 2D surface that was "converted" to the 3D surface.

```
BOOL CSurface3d::AttachZbuffer(LPDIRECTDRAWSURFACE p2dFns)
{
    // Attach the Z-buffer to the 2dSurface so D3D will find it
    p2dFns->AddAttachedSurface(m_pZSurfFns);
    // set internal state and return
    m_bIsZEnabled = TRUE;
    return TRUE;
}
```

One more task: we've got to tell the renderer to use Z-Buffering. As usual, we communicate with the renderer through an Execute Buffer. Here are the state variables that we've got to toggle to let the renderer know about Z-Buffering:

Turn on Z-Buffering by setting the D3DRENDERSTATE_ZENABLE to TRUE.

```
// Don't forget to increase the size of execute buffer needed
OP_STATE_RENDER(1, lpTmp);
    STATE_DATA(D3DRENDERSTATE_ZENABLE, m_bIsZEnabled, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZWRITEENABLE, m_bIsZEnabled, lpBuffer);
    STATE_DATA(D3DRENDERSTATE_ZFUNC, D3DCMP_LESSEQUAL, lpBuffer);
```

Although their default values are adequate for our purposes, you may also want to look at the states of the additional two Z-Buffering control states:

D3DRENDERSTATE_ZWRITEENABLE	Default is TRUE. If set to FALSE, the renderer will continue to examine the Z-Buffer while rendering (as specified by _ZENABLE); but it will not update the Z-Buffer with new z-values.
D3DRENDERSTATE_ZFUNC	Default is D3DCMP_LESSEQUAL. With the z-comparison function, you can change the way the renderer interprets z-distance. For instance, you can reverse z-ordering by setting the state to D3DCMP_GREATEREQUAL.

Refer to Direct3D documentation for more information.

Run the demo for this chapter; check both the Z-Buffer option and the intersecting triangles option. Play around with toggling Z-Buffering on and off and see how the intersecting portion of the triangle 2 is and isn't rendered. You should also notice that the render order of the triangles changes as you switch between Z-Buffered and non-Z-Buffered rendering.

It's time to conclude this chapter. We proclaim you to be proficient with Direct3D—that is, as long as you actually worked with the code!

WHAT HAVE YOU LEARNED?

Wow! Our triangle has come a long way, baby! We started the chapter with a plain old triangle with a single color that left triangle trails. Here's what we have accomplished since then:

- The first thing we covered was removing those annoying triangle trails through Direct3D backgrounds. Along the way we learned about and worked with Direct3D materials.
- Next we looked at how Direct3D dealt with coloring and shading triangles. Then we played around with vertex colors to see how Gouraud shading looked.
- Gouraud shading produced banding artifacts, so we learned about render states and improved the quality of Gouraud shading by changing a render state to turn on dithering.
- With ever increasing confidence in navigating through Direct3D, we tackled texture mapping—"converting" a DirectDraw surface to a texture object, and rendering triangles with our texture map.
- Finally, we tried our hand at some true 3D rendering—turning on Z-Buffering and even rendering intersecting triangles correctly.

We have come a long way from where we started. We're sure you're pretty handy with Direct3D programming by now. Let's turn our attention to performance in the next chapter.

CHAPTER 16

Understanding and Enhancing Direct3D Performance

WHY READ THIS CHAPTER?

Previous chapters showed rendering a simple triangle with Direct3D, adding various shading options, and then adding texture mapping to the triangle. But we were mainly concerned with getting the basic application running, so we didn't really pay much attention to how fast it ran. In this chapter we'll take our measuring microscope out and measure the performance of our previous examples. Then we'll look at ways to accelerate these samples.

By working through this chapter you will

- get a feel for the performance of various rendering options with the RGB model driver,
- learn how to use the Ramp model driver to get better performance, and
- measure the results of rendering using the Ramp model driver.

16.1 How Fast Does Our Triangle Run?

We've been drawing triangles with various options through Direct3D. So far we've focused on getting things to work and on exploring possibilities. But for serious multimedia application development, we've also got to focus on performance.

Direct3D's Immediate mode API is a very low level API. But it was designed this way to offer high performance. In which case, why measure performance? Shouldn't the fact that we're using Immediate mode be sufficient? Sadly, no. Not all performance paths are equal (some are more equal than others).

Let's measure the performance of our triangle on Direct3D and then look at performance enhancement opportunities. But first a word about our measurements:

- All measurements were taken on our base platform described in Section 2.5. Results will definitely vary on different platforms or even on the same platform with different display configurations.

- We have included the timing application and its source code on the CD. Use it to profile your platform and see how it performs.
- We have separated the timing source code from the source for the basic demos, to simplify reading the base code and to give you a performance tool to measure various configurations.

16.1.1 Stages of Rendering Our Triangle

Let's start by looking at what it takes to render a scene. We have broken the scene rendering into the following stages:

- *Init.* This stage occurs once while the entire application is being initialized. We put all our one-time initialization activities into this stage. This stage is typically not time critical, and we do not measure this stage.
- *Prepare Scene.* This stage occurs at the start of every scene render. We invoke *IDirect3DDevice::BeginScene()* in this stage. If we used Z-Buffers, we would typically clear them in this stage (prior to objects being rendered).
- *Draw background.* If a background was created, then we would invoke the *Background::Blt()* function in this stage, which translates to *IDirect3DViewport::Clear(...target...)* in Direct3D. We are not measuring a background for our base case triangle.
- *Edit Execute Buffer.* Typically, objects in a 3D scene are moving around. (If we were only going to draw a stationary object, then we might as well Blt a bitmap.) This is the stage when we edit the Execute Buffer. We then need to transfer the edited data to the Direct3D driver, using the hopefully familiar Lock, Copy, and Unlock sequence.
- *Set Execute Buffer.* In the code from the previous chapters, we saw that after transferring Execute Buffer data, we described the makeup of the new Execute Buffer to the Direct3D device.
- *Execute.* Here's where we get the device to execute our Execute Buffer (or Buffers, if we had many).
- *End Scene.* This stage occurs at the end of every stage render. First we invoke *IDirect3DDevice::EndScene().* Next we set our palette and refresh the window with the newly rendered scene.

16.1.2 Measuring the Rendering Stages of Our Triangle

Table 16-1 measures the time taken in the various rendering stages to render our triangle. We are using the Direct3D RGB color model driver with no hardware acceleration. (The default Rendering state is Gouraud shaded, solid fills with specularity enabled, and dithering disabled.)

TABLE 16-1 Timing Render Stages for Our Base Triangle

Stage	Time (milliseconds)	
	625 × 128	16 × 5000
Prepare scene	0.0	0.0
Edit Execute Buffer	10.9	0.4
Set Execute Buffer	3.3	0.1
Execute	75.1	60.6
End scene	7.5	7.5
TOTAL	96.6	68.5

We measured two scenes: The first scene had 625 small triangles. For this initial test, we invoked Blt for every triangle. All the triangles were of the same shape—with reference vertices of (0,0), (16, 16), and (0,16)—for a size of 128 pixels per triangle and 80,000 pixels per scene. We chose this configuration based on experience as representative of medium complexity 3D applications.

The second scene had 16 large triangles. Again, we used triangles of the same shape and invoked Blt for each one. Reference vertices were (0,0), (100, 100), and (0,100)—for triangles of 5,000 pixels each and 80,000 pixels per scene. We measured this configuration to demonstrate the impact that polygon size can have on render performance.

We varied the positions of the triangles in each scene to study the effect of alignment. The tabulated values are the averaged results.

Following are some observations on the results:

- Large triangles rendered significantly faster than small triangles. Even though the total pixels in both scenes were the same, the render time for the second scene was much faster than the first scene. This indicates that you would increase performance by using, wherever possible, larger triangles instead of a bunch of smaller triangles.

- The cost of invoking *BeginScene()* and *EndScene()* in this example is imperceptible.
- Editing the Execute Buffer to reposition vertices consumes a significant amount of time in scene 1. (The cost is not noticeable in scene 2, since there aren't too many triangles in that scene.)

16.1.3 Trimming Some Fat from the Rendering Stages

Let's make changes to our Render stages to see if there's some easy performance pickin's to be had:

- *Set Execute Buffer (1).* If our only changes to an Execute Buffer are data related, and we do not change the makeup of the Execute Buffer, then do we really need to "redescribe" the Execute Buffer to the Direct3D device? The Direct3D documentation does not specify what's correct behavior. We ran tests and found that we *do not* need to "redescribe" if only the data changes. (In fact, as long as the four fields of the `D3DEXECUTEDATA` structure do not change, then we can even change instruction opcodes and operands without "re-describing" the Execute Buffer.)*
- *Set Execute Buffer (2).* If we only wanted to change a couple of vertices, do we really have to recopy the entire buffer to the Direct3D device? In other words, can we see our previous data with the pointer returned by *Lock()*? We ran tests and found that *yes, indeed,* we do get access to our previous data, and we can edit in place if we wish. *
- *End Scene.* Our initial code reset the palette on every scene render. Again, we ran some tests and found that the Direct3D RGB color model driver does not change the palette from frame to frame. We rewrote our code to set up the palette only when our application gets focus and removed this work from the End Scene stage.*
- *Execute.* Our initial code invoked *Blt()* on each triangle. We were executing an Execute Buffer with only one triangle. We rewrote this code to execute all our triangles (625 or 16) via single Execute Buffers (using only one buffer per list of triangles).*

* You may wish to reverify the results and rerun these tests with any hardware accelerator drivers you choose to use.

Table 16-2 lists new measurements based on the edited code:

TABLE 16-2 Trimming Some Fat from the Render Stages

Stage	Original (milliseconds)		Long Buffer (milliseconds)		Edit in Place (milliseconds)	
	625 × 128	16 × 5000	625 × 128	16 × 5000	625 × 128	16 × 5000
Prepare scene	0.0	0.0	0.0	0.0	0.0	0.0
Edit Execute Buffer	10.9	0.4	1.1*	0.0*	10.4	0.3
Set Execute Buffer	3.3	0.1	0.0	0.0	0.0	0.0
Execute	75.1	60.6	72.4	60.7	75.4	60.7
End scene	7.5	7.5	7.1	7.1	7.1	7.1
TOTAL	96.7	68.6	*	*	92.9	68.1

*We used pre-initialized Execute Buffers, while testing Long Execute Buffers, and we have listed only the time taken to copy the buffers to the 3D device space. For a fair apples-to-apples comparison, you would need to add the time taken to edit the vertex positions in the pre-initialized Execute Buffers.

Following are some conclusions based on our code rearrangement:

- Not "redescribing" the Execute Buffer and using Long Execute Buffers saved 16 milliseconds in scene 1. While the savings are low relative to the Execute cost, we will find that the savings are significant when we find faster Execute methods (as we will see later in this chapter).
- None of the changes produced any significant tangible benefit for scene 2, which indicates that overhead in Direct3D has been minimized, and it would only become significant over a large number of invocations.
- Not setting the palette only saved 0.3 milliseconds. This is a useful measurement to remember, since you can retain the code to constantly change palettes in case a Direct3D driver does change palettes frequently.

16.2 Measuring Shading Options

Over the course of the previous two chapters, we have rendered our triangle with a variety of options, including Gouraud shading, flat shading, dithering, texture mapping, and Z-Buffering. Let's look at how these options affect our render performance.

16.2.1 Measuring the Performance of Shading Options in Our Triangle

Table 16-3 measures the time taken to draw triangles with various shading options. We are using the Direct3D RGB color model driver with no hardware acceleration. We are only measuring one of the two scenes from the

previous tests—the scene with sixteen triangles of 5,000 pixels each. Once again, we drew the triangles at various alignments and have tabulated averaged results. The timings listed are only those taken to render the sprites. Redrawing backgrounds and refreshing the screen are additional costs.

TABLE 16-3 Timing Sixteen Triangles with the Direct3D RGB Color Model Driver

Rendering Option	Time	Megapixels/Second
Gouraud	55.3 milliseconds	1.45
Gouraud (with constant colors)	55.3 milliseconds	1.45
Flat shaded	55.3 milliseconds	1.45
Gouraud and dither	55.3 milliseconds	1.45

Two glaring observations leap out at us:

- Flat shading *does not perform better* than Gouraud shading. You would expect that not computing shaded colors would result in better performance. But it doesn't.
- Enabling dithering *does not reduce* rendering performance. This is a rare occasion—you can add an improvement at no performance cost.

Since varying shading options didn't make any difference whatsoever to rendering performance, we decided to see whether this constant performance persisted after we added some specular highlights.

Table 16-4 compares the performance of the shading options with and without specular highlights. We found that adding specular highlights cost us a 9 percent performance penalty.

TABLE 16-4 Measuring the Impact of Adding Specular Highlights

Rendering Option	Without Specular Highlights	With Specular Highlights
Gouraud	55.3 milliseconds	60.3 milliseconds
Gouraud (with constant colors)	55.3 milliseconds	60.3 milliseconds
Flat shaded	55.3 milliseconds	60.2 milliseconds
Gouraud and dither	55.3 milliseconds	60.0 milliseconds

Note that all these measurements were taken with the RGB color model driver rendering to an 8 bpp palletized target surface. Since shading options and specular highlighting modify individual RGB components, you can bet that performance on 16-, 24-, and 32-bpp targets will be very different.

EXTRA CREDIT

Our performance application for this chapter will only work with 8 bpp palletized targets. To motivate you to work with the source code, we've left supporting other target modes as an extra credit exercise. Go on! It's fairly simple.

16.2.2 Measuring the Performance of Texture-Mapping in Our Triangle

Next we move on to measuring texture-mapped triangles. Remember from our discussion of rendering options in Section 15.3.3 that

- texture mapping can be disabled, but shading cannot;
- the default texture mapping mode combines texels with shading values; and
- texture mapping needs to be set to Copy mode to turn shading off.

Table 16-5 tabulates measurements of rendering our triangles with a texture map of 64 × 32 pixels. For good measure, we've included the impact of adding specular highlights to our texture-mapped triangles.

TABLE 16-5 Texture-Mapped Triangle with and without Specular Highlights

Rendering Option	Without Specular Highlights		With Specular Highlights	
	(milliseconds)	(megapixels/ second)	(milliseconds)	(megapixels/ second)
Texture and Gouraud	62.5	1.28	68.2	1.18
Texture and Flat Shaded	62.5	1.28	67.9	1.18
Texture and Gouraud and Dither	62.7	1.28	67.9	1.18
Copy Mode	14.4	5.56	14.4	5.56

Wow! CopyMode gives us more than *four times better performance* over Modulated mode texture mapping (with shading options turned on). An obvious conclusion is that making an effort to use pre-lit textures will definitely reap significant performance benefits. You could even use texture mapping with flat-shaded textures to get high-performance flat shading.

16.2.3 Adding a Z-Buffer to the Recipe

There was one other option that we looked at in the previous chapter: Z-Buffering. Without Z-Buffers, your application must send triangles to the renderer sorted in back-to-front order, and your application must also subdivide intersecting triangles.

With Z-Buffers, the driver correctly renders both unsorted and intersecting triangles, saving your application the cost of sorting and subdividing. So how much does Z-Buffered rendering cost?

TABLE 16-6 Rendering Cost of Z-Buffering (Direct3D RGB Driver)

Rendering Option	Without Z-Buffer	With Z-Buffer
Gouraud	55.3 milliseconds	60.8 milliseconds
Gouraud and Specular	60.3 milliseconds	66.6 milliseconds
Texture Map and Gouraud	62.5 milliseconds	69.5 milliseconds
Texture Map CopyMode	14.4 milliseconds	30.1 milliseconds

Table 16-6 compares the cost of rendering our triangle with and without Z-Buffering. And the verdicts are in:

- Rendering with Z-Buffering *is* more expensive than rendering without. No surprises here, since Z-Buffering increases memory traffic.
- The cost of Z-Buffering is only in the 10 percent range for the expensive rendering options, suggesting that Z-Buffering is a viable option in cases where performance isn't supercritical.
- Rendering with Z-Buffering *severely impacts* CopyMode. This is unfortunate, since CopyMode is our fastest mode. Therefore, for high-performance 3D, you really need to measure the cost of sorting and segmenting triangles and compare this cost against the cost of Z-Buffering.

Our Timing Application gives you an opportunity to measure how much Z-Buffering will cost you. We hope that measurements (or measurement tools) like these will help you decide whether the performance of Z-Buffering is adequate or whether it's worth investing the effort on the non-Z-Buffered path.

In addition to rendering options, other factors also affect rendering performance. Both the size and the shape of a triangle can have effects on performance. Similarly, with texture-mapped rendering, the size and shape of textures also have an impact on rendering performance.

16.2.4 Getting Perspective: Comparing 3D (RGB Mode) to 2D

OK, we've bandied around numbers from 14 milliseconds to 70 milliseconds. But we've been comparing various options within the 3D realm. How do we know whether these numbers are acceptable? What if we compare our 3D rendering throughput against the 2D throughput we have seen when we worked with sprites in Part II?

Of course, 3D rendering will probably be slower. But how much slower—let's get some perspective. Table 16-7 compares Sprite render times from Part II to our measurements from this chapter. Note that the sprites in Part II were also about 5,000 pixels, and each was rendered sixteen times. In terms of speed, the comparison puts our best 3D render mode on an even par with spriting through GDI. Talk about backwards! We need to get better.

TABLE 16-7 Comparing 3D Throughput to 2D Sprite Throughput

Rendering Path	Time	Megapixels/Second
GDI	14.2 milliseconds	5.96*
CSpriteCCode	11.9 milliseconds	7.12*
CSpriteP5	~1.5 milliseconds	56.5*
CSpriteGrfx	~1.5 milliseconds	56.5*
CTriangle3D (Gouraud)	55.3 milliseconds†	1.45
CTriangleTex(Copy mode)	14.4 milliseconds†	5.56

* Some pixels are transparent, and throughput is somewhat less than what these figures indicate.

† Measurements will take longer when CTriangleTex is rendered directly to a triple-buffered video memory surface.

Why is it so important to get better performance? Let's assume

- that we'd like a real-time feel with a frame rate of around 30 fps. At this frame rate we get 33 milliseconds per frame to work with; typically about

half this time is consumed on peripheral activities such as responding to the user and carrying out geometry and lighting computations and audio and 2D graphics activities;

- a best-case scenario where the graphics device has enough memory to support triple buffering, and screen refreshes are occurring at no cost; and
- that we'd like our application to occupy a screen resolution of about 640 × 480; a 640 × 480 background gets drawn to the triple-buffered video surface at a cost of 5 milliseconds.

We're left with about 12 milliseconds for 3D rendering. Our best performance mode will render in that time about 65,000 pixels to a system memory buffer. Even if no pixel was rendered more than once, we would be painting an area of about 320 × 200 with 3D pixels—less than one quarter of the screen area. With more realistic assumptions that pixels are touched about 1.5 times on average, our 3D coverage reduces to about 240 × 180—a postage-stamp-sized field of action.

Better performance means covering more of the screen area with 3D pixels or alternately it means being able to run richer multimedia applications with, for example, full-motion video being used as texture map sources. So let's look at Direct3D's high-performance option—the Ramp (Mono) color model driver—to see if we can get better 3D rendering performance on a PC.

16.3 Improving Performance Using the Ramp Driver

We've mentioned before that the DirectX SDK ships with two implementations: an RGB color model driver and a Ramp (Mono) color model driver. The RGB driver offers truer color quality but runs more slowly. The Ramp (Mono) driver makes color approximations that degrade overall quality, but it offers higher performance. Let's start using the Ramp color model driver.

16.3.1 Loading the Ramp Color Model Driver

Right after we started using Direct3D in Section 14.4.2, we looked at code to enumerate available device drivers. With the code listed there we selected a driver based on a selection criteria that we passed down. Among the possible selection criteria was `USE_RAMP`. At the initialization level, loading the Ramp driver is simple, as shown in the code.