Chapter 24

A DirectX 5 Update

Why Read This Chapter?

Time waits for no person. While we were putting our book together, Microsoft was working on the next generation of DirectX—DirectX foundation 5. In all likelihood, foundation 5 will become available before this book does.

For the most part, DirectX foundation 5 is similar to DirectX 3. Read the beginning of this chapter for coverage on some notable changes. The most significant change in DirectX foundation 5 is the addition of a new DrawPrimitive API to program Direct3D's Immediate mode. The DrawPrimitive API is now the preferred API, though the Execute Buffer API is still usable.

In this chapter, we'll convert some of our earlier render triangle code to use the DrawPrimitive API. By the time you have worked through this chapter, you will

- get a feel for some of the changes in DirectX foundation 5, and
- learn how to render a texture-mapped triangle using DrawPrimitives.

24.1 DirectX 5

What's new with DirectX 5? Well, right away you'll notice that it's called DirectX *foundation* 5. The components in the DirectX kit have grown in the three years that DirectX has been in existence. Microsoft has separated the components into layers.

The *foundation* layer makes up the bottom-most layer of the DirectX kit. It contains the lowest-level interfaces, including DirectDraw, DirectSound,

and Direct3D Immediate mode. At a slightly higher layer is DirectX Media, which includes Direct3D Retained mode, DirectAnimation, and DirectShow (previously ActiveMovie).

The low-level foundation layer offers interfaces that are very close to the hardware; while the higher-level Media interfaces offer higher-level abstractions and sit on top of (make use of) the foundation-level interfaces.

Digging deeper, there are several other changes in DirectX that are of interest to us. Let's look at some changes to the DirectDraw and Direct3D portions of DirectX.

24.2 DirectDraw Changes in DirectX 5

Here are some of the changes to DirectDraw in the DirectX foundation 5 release:

- It can control whether the Hardware Emulation Layer (HEL) is used on surfaces.
- Palette creation and setup are slightly different.
- Offscreen surfaces can be wider than the Primary surface.
- New flags have been added to support accelerated graphics port (AGP)–based memory.

24.2.1 Controlling the Use of the DirectDraw Hardware Emulation Layer

The following code box shows the standard *DirectDrawCreate()* call that is the starting point of all DirectDraw functionality. The parameter list seems unchanged. The first parameter is a GUID that lets us specify which DirectDraw driver we want to use. The newly created DirectDraw object is returned in the space we provide in the second parameter.

```
HRESULT DirectDrawCreate(
  GUID FAR *lpGUID,
  LPDIRECTDRAW FAR *lplpDD,
  IUnknown FAR *pUnkOuter);
```

With previous versions of DirectX we could set the GUID to `NULL` to indicate that we wanted the DirectDraw driver connected to the primary display. With DirectX 5, we get additional control. The GUID can also be set to `DDCREATE_EMULATIONONLY` or to `DDCREATE_HARDWAREONLY`.

With the GUID set to DDCREATE_EMULATIONONLY, the DirectDraw object will use emulation for all features; it will not take advantage of any hardware acceleration. With the GUID set to DDCREATE_HARDWAREONLY, the DirectDraw object will never emulate features not supported by the hardware. Attempts to call methods that require unsupported features will fail, returning DDERR_UNSUPPORTED.

24.2.2 Changes in Palette Setup

The DDCAPS_INITIALIZE is now obsolete. The implication is that you can no longer create empty DirectDraw palettes and then set their entries later. The *IDirectDraw2::CreatePalette()* call expects a valid palette-entry array and always initializes from it.

24.2.3 Creating Wide Offscreen Surfaces

DirectDraw now supports Offscreen surfaces that are wider than the Primary surface. You can create surfaces as wide as you need, as long as the DirectDraw driver indicates that it supports wide surfaces. (Look for DDCAPS2_WIDESURFACES returned from *IDirectDraw2::GetCaps()*.)

If you attempt to create a wide surface in video memory when the DDCAPS2_WIDESURFACES flag isn't present, the attempt will fail, and DDERR_INVALIDPARAMS will be returned. Wide surfaces are always supported for system memory surfaces.

24.2.4 AGP Support

DirectDraw now has support for AGP-based memory. Accelerated graphics port (AGP) memory is memory specially arranged for fast reads by the graphics controller. Whenever the graphics processor reads from AGP memory, caching at the processor needs to be partially disabled to make sure that the data is accurate—which means that the DirectDraw layer must do some management tasks whenever you ask for AGP memory.

In addition, AGP memory is system memory that tries to look like video memory. The disguise is successful for graphics processor reads. But as of yet, graphics processor writes to AGP memory are significantly slower than equivalent writes to real video memory. Some graphics processors cannot tolerate slow response times, and therefore a distinction must be made between real video memory and virtual video memory.

At the *IDirectDraw::CreateSurface()* level, DirectDraw supports AGP by adding DDSCAPS_LOCALVIDMEM and DDSCAPS_NONLOCALVIDMEM flags. As the names would suggest, DDSCAPS_LOCALVIDMEM indicates that the surface must exist (exists) in true, local video memory, and DDSCAPS_NONLOCALVIDMEM indicates that the surface must exist (exists) in AGP video memory.

To let you find out whether the graphics processor can Blt among the various memory types, several new caps fields have been added into the DDCAPS structure. Some examples are

```
DWORD dwNLVBCaps;       // nonlocal-to-local video memory Blt caps
DWORD dwSVBCaps;        // system-to-video Blt-related caps
DWORD dwVSBCaps;        // video-to-system Blt-related caps
DWORD dwSSBCaps;        // system-to-system Blt-related caps
```

24.3 Direct3D Changes in DirectX 5

The significant changes in the DirectX foundation 5 release are in the Direct3D component. Among them, we'd like to point out the following:

- MMX optimizations are explicitly invoked through a separate driver.
- The Execute Buffer API is replaced by DrawPrimitive API.
- A new IDirect3DViewport2 object extends the original object with a simplified D3DVIEWPORT2 structure.

24.3.1 A Separate Direct3D MMX Technology Driver

With DirectX 3, either MMX technology or non–MMX technology versions of Direct3D drivers were loaded on the system based on an install time option presented to the user. This model presented two problems:

- the choice of which driver to install was left to the user; and
- once the user installed one set of drivers, the other set of drivers could not be accessed, even if the other set would work faster.

DirectX 5 has distinctly separated the two drivers. It always installs the non–MMX technology driver. Additionally, if the target platform has an MMX technology processor, DirectX 5 will add the MMX technology driver automatically without user intervention.

As a result of the new model, the MMX technology driver must be explicitly loaded. *IDirect3D::EnumDevices()* will enumerate to the specified callback, any available MMX technology driver as an additional option. (See section 14.4.2 for a discussion of enumerating devices.) Unfortunately, there is no flag defined to identify an MMX technology driver; if you want to identify MMX technology enhancements during the enumeration, you will have to search for the key words "MMX technology" in the driver name; alternately you can arrange to use hard-coded GUIDs, which you have already determined to have MMX technology enhancements.

24.3.2 DrawPrimitive API

DirectX 5 now has a new Direct3D API called the DrawPrimitive API. The old ExecuteBuffer model suited the needs of a certain class of applications, but for certain other applications, it was very hard to program and use. In addition, the Execute Buffer API was hard to debug. The new DrawPrimitive API was designed for easier programming and debugging.

The DrawPrimitive API is accessible through newly defined IDirect3D2 and IDirect3DDevice2 objects. The new IDirect3DDevice2 object is *not* an extension of the old IDirect3DDevice object, and the new object does not inherently support the old API. If you want to use the old API, you will need to create the old object, or you will need to *QueryInterface()* the old object from the new one.

Although IDirect3D2 objects are created as before (see section 14.4.1), the new IDirect3DDevice2 is created with the *IDirect3D2::CreateDevice()* call.

```
HRESULT CreateDevice(
  REFCLSID               rclsid,
  LPDIRECTDRAWSURFACE    lpDDS,
  LPDIRECT3DDEVICE2   *lplpD3DDevice2
);
```

When you call *IDirect3D2::CreateDevice()*, you create a device object that is separate from a DirectDraw surface object. In fact, you specify a target DirectDraw surface as a parameter to the *CreateDevice()* call.

As we mentioned before, IDirect3DDevice2 supports only the new DrawPrimitive methods—there are no *IDirect3DDevice2::ExecuteBuffer* functions. You can still use the Execute Buffer model by retrieving an

IDirect3DDevice object through *IDirect3DDevice2::QueryInterface()*. Also Execute Buffers are still available by directly creating an IDirect3DDevice.

24.3.3 New IDirect3DViewport2 Object

IDirect3DViewport2 is identical to IDirect3DViewport except there are two new methods: *GetViewport2()* and *SetViewport2()*. These calls use a new `D3DVIEWPORT2` structure. This structure allows for a closer match between window size and viewport size than is possible with the `D3DVIEWPORT` structure.

24.4 Programming Direct3D's Immediate Mode with DirectX 5

Let's render a texture-mapped triangle with DrawPrimitives. Why go straight for texture mapping? Well, the DrawPrimitive API also affects changing states in the renderer. We picked an example that shows the changes to both triangle descriptions and state modifications.

24.4.1 Original Execute Buffer Code for Texture-Mapped Triangle

This is our original Execute Buffer code for a texture-mapped triangle:

```
BOOL CTriangleRamp::Init(LPDIRECT3D pD3D, UINT nRes)
{
```

Create a material to "set" palette entries for color.

```
  pD3D->CreateMaterial(&m_pMaterialFns, NULL);
  m_MaterialDesc.dcvDiffuse.dvR = D3DVALUE(1.00);
  m_MaterialDesc.dcvDiffuse.dvG = D3DVALUE(1.00);
  m_MaterialDesc.dcvDiffuse.dvB = D3DVALUE(1.00);
  m_MaterialDesc.hTexture = NULL;
  m_MaterialDesc.dwRampSize = 16;
  m_pMaterialFns->SetMaterial(&m_MaterialDesc);
  m_pMaterialFns->GetHandle(p3dFns, &m_hMaterial);
```

1. Allocate system memory space for an Execute Buffer.

```
  #define nTRIS  1
  #define nVERTS nTRIS*3
   m_sztEx = sizeof(D3DTLVERTEX) * nVERTS;
   m_sztEx += sizeof(D3DINSTRUCTION) * 5;
   m_sztEx += sizeof(D3DSTATE) * 2;
   m_sztEx += sizeof(D3DPROCESSVERTICES);
   m_sztEx += sizeof(D3DTRIANGLE) * nTRIS;
   m_pSysExBuffer = new BYTE [m_sztEx];
   memset(m_pSysExBuffer, 0, m_sztEx);
```

2. Set up vertices in Execute Buffer.

```
D3DTLVERTEX *aVerts = (D3DTLVERTEX *)m_pSysExBuffer;
setupVertices(nTRIS, aVerts);
```

3. Set up state instructions in Execute Buffer.

```
DWORD dwStart = sizeof(D3DTLVERTEX) * nVERTS;
LPVOID pTmp = (LPVOID)(m_pSysExBuffer + dwStart);
OP_STATE_LIGHT(1, pTmp);
   STATE_DATA(D3DLIGHTSTATE_MATERIAL,m_hMaterial,pTmp);
OP_STATE_RENDER(2,pTmp);
   STATE_DATA(D3DRENDERSTATE_SHADEMODE,D3DSHADE_GOURAUD,pTmp);
   state_data(d3drenderstate_texturehandle, m_hMaterial,pTmp);
OP_PROCESS_VERTICES(1, pTmp);
   PROCESSVERTICES_DATA(D3DPROCESSVERTICES_COPY,0,nVERTS,pTmp);
```

4. Set up triangle list in Execute Buffer.

```
OP_TRIANGLE_LIST(nTRIS, pTmp);
for (i=0; i<nTRIS; i++) {
   ((LPD3DTRIANGLE)pTmp)->v1 = i*3+0;
   ((LPD3DTRIANGLE)pTmp)->v2 = i*3+1;
   ((LPD3DTRIANGLE)pTmp)->v3 = i*3+2;
   ((LPD3DTRIANGLE)pTmp)->wFlags = 0;
   lpTmp = ((char*)pTmp) + sizeof(D3DTRIANGLE);
}
```

5. Close instruction list in Execute Buffer and transfer to 3D driver.

```
OP_EXIT(pTmp);
DWORD dwLth = (LPBYTE)pTmp - m_pSysExBuffer - dwStart;
m_ExDesc.dwFlags = D3DDEB_BUFSIZE;
m_ExDesc.dwBufferSize = m_sztEx;
p3dFns->CreateExecuteBuffer(&m_ExDesc,&m_pExBufFns,NULL);
m_pExBufFns->Lock(&m_ExDesc);
lpTmp = (LPBYTE)m_ExDesc.lpData;
memcpy(lpTmp, m_pSysExBuffer, lpInsEnd-m_pSysExBuffer);
m_pExBufFns->Unlock();
m_ExData.dwVertexOffset = 0;
m_ExData.dwVertexCount = 3;
m_ExData.dwInstructionOffset = lpInsStart-m_pSysExBuffer;
m_ExData.dwInstructionLength = dwLth;
m_pExBufFns->SetExecuteData(&m_ExData);
```

24.4.2 New DrawPrimitive Code for Texture-Mapped Triangle

With Execute Buffers, we rendered a triangle by

- calculating and allocating the space required for the Execute Buffer;
- setting up vertices in the Execute Buffer;
- setting up state instructions in the Execute Buffer;
- connecting vertices into a triangle list to describe the triangle;
- closing the Execute Buffer;

- describing the Execute Buffer to the 3D driver via a descriptor;
- copying the Execute Buffer to the 3D driver (in between lock/unlock); and
- executing the Execute Buffer.

With the new DrawPrimitive API

- The concept of Execute Buffers no longer exists. That means that steps 1 and 5–8 are no longer needed.
- A triangle list is described by the order of the vertices. Every set of three vertices in the vertex list describes a triangle. That means that step 4 is not needed, and step 2 might need to be modified for in-order vertices.
- Render states, rather than being inserted into the Execute Buffer, are now set with explicit function calls. That means that step 3 is modified. This new method of using explicit function calls, as we will soon see, is much easier to work with.
- Last, step 8, the step of executing the Execute Buffer, is now changed to an explicit DrawPrimitive call that immediately renders the required triangle. The Process Vertices operation from the previous Execute Buffer model is now obsolete, and its purpose has been folded into the DrawPrimitive call.

So now the new set of steps is

- set States (LightStates and RenderStates) with explicit function calls;
- set up triangle vertex list—every triangle is described by a set of three vertices; and
- cause the triangles to be drawn with an explicit DrawPrimitive function call.

Here is our code converted for use with the DrawPrimitive API:

```
BOOL CTriangleRamp::Init(LPDIRECT3D2 pD3D, UINT nRes)
{
    Use standard code to set material and to "set" palette entries for color.
  pD3D->CreateMaterial(&m_pMaterialFns, NULL);
  m_MaterialDesc.dcvDiffuse.dvR = D3DVALUE(1.00);
  m_MaterialDesc.dcvDiffuse.dvG = D3DVALUE(1.00);
  m_MaterialDesc.dcvDiffuse.dvB = D3DVALUE(1.00);
  m_MaterialDesc.hTexture = NULL;
  m_MaterialDesc.dwRampSize = 16;
```

```
m_pMaterialFns->SetMaterial(&m_MaterialDesc);
m_pMaterialFns->GetHandle(p3dFns, &m_hMat);
```

Initialize in-order list of vertices.

```
#define nTRIS  1
#define nVERTS nTRIS*3
m_sztVerts = sizeof(D3DTLVERTEX) * nVERTS;
m_pVerts = (D3DTLVERTEX *)(new BYTE [m_sztVerts]);
memset(m_pSysExBuffer, 0, m_sztEx);
setupVertices(nTRIS, m_pVerts);
```

Set states.

```
err = p3dFns->SetLightState(D3DLIGHTSTATE_MATERIAL, m_hMat);
macExitIfD3DError(err, FALSE);
p3dFns->SetRenderState(D3DRENDERSTATE_TEXTUREHANDLE,m_hTex);
```

SetLightState() and *Set RenderState()* are new member functions of the IDirect3DDevice2 class. With these calls, we can set each state explicitly and get immediate error feedback. In addition, with the new approach, we don't have to specify how many states we're setting—previously, a common opportunity for introducing bugs.

Render triangle list.

```
DrawPrimitive(D3DPT_TRIANGLELIST, D3DVT_TLVERTEX,
                m_pVerts, nVERTS, D3DDP_DONOTCLIP | D3DDP_WAIT);
```

The first parameter to DrawPrimitive specifies what kind of primitive to draw. Options with DirectX 5 are `_TRIANGLELIST`, `_LINESTRIP`, `_TRIANGLESTRIP`, `_TRIANGLEFAN`, `_POINTLIST`, AND `_LINELIST`.

The second parameter specifies what kind of vertices make up the primitives, and some options are `D3DVT_VERTEX`, `D3DVT_LVERTEX`, or `D3DVT_LVERTEX`. We used to pass this information with the `OP_PROCESSVERTICES` instruction.

The last parameter to *DrawPrimitive()* is a flags parameter. Among the DrawPrimitive flags is the familiar `_WAIT` flag that controls asynchronous drawing. Other flags have been defined to pass hints to the D3D driver; refer to the D3D documents for more information.

```
}
```

24.5 Demo Time

We have compiled the code for this chapter and modified our menu program for the new option. Now let's run the program. The triangle you see is being rendered with the new DrawPrimitive API.

What Have You Learned?

We started out by looking at those changes in DirectX 5 that affect our previous DirectDraw and Direct3D work, and then we finished the chapter with hands-on work with the most significant DirectX 5 feature—the DrawPrimitive API. You now have a working example of rendering a texture-mapped triangle with the DrawPrimitive API.