Chapter 2

Direct3D

"We may live without architecture, and worship without her, but we cannot remember without her."

John Ruskin: *The Seven Lamps of Architecture*, VI

2.1 Overview

This chapter begins with a description of the general architecture of Direct3D. We discuss how Direct3D relates to GDI under Windows, and then introduce the abstractions used by Direct3D: devices, swap chains, surfaces, and resources. Before getting into the details of Direct3D, we show a complete Direct3D application that draws a single triangle and explain its various parts.

We next describe the `IDirect3D9` interface. This interface exposes the graphics devices present in the system. We summarize the interface methods before elaborating on them in the following sections.

When a graphics application initializes itself, it first probes the system devices and selects an appropriate device for the display needs of the application. This process is referred to as "device enumeration." `IDirect3D9` makes enumeration easy by allowing us to examine all the adapters present in the system for appropriate device and display mode support.

Each monitor is driven by an adapter, which supports a number of video display modes varying in screen dimensions and refresh rate. Each adapter can expose hardware and software device objects that can operate in windowed or exclusive mode.

An application enumerates the devices available on each adapter, examining their capabilities and supported display modes to find an acceptable device. After a device has been found to have suitable capabilities, the application checks to see if the device supports the required render target formats, resource formats and multisampling support needed by the application.

A Direct3D application can enumerate devices on the default adapter or on all adapters. As multiple monitor systems become more common, an application can take advantage of the additional display space afforded by additional monitors. We present a guideline for multiple monitor applications.

2.2 A Minimal Direct3D Application

Before we get into the details of a Direct3D application, lets start with a minimal but complete example of a Direct3D application. Such a program is given in listing 2.1.

The program draws a scene consisting of a single triangle aligned with the screen (i.e. parallel to the plane of the screen). This program produces output that could also be achieved with GDI, but as we will see in the following chapters Direct3D can achieve much more than GDI at much faster speeds.

Drawing a single triangle that directly faces the viewer could be considered a "two-dimensional" application. Direct3D exposes powerful graphics hardware that can accelerate 2D applications just as well as 3D applications. Many times 2D applications are 3D applications in disguise – they draw 3D primitives that are aligned with the screen.

This application, while minimal, goes through all the operations a typical Direct3D application will perform during its lifetime. For this minimal example, the operations are simplified, but the basic structure remains the same.

this program registers a window class procedure and creates a window like most Win32 applications. It then acquires the necessary interfaces for using Direct3D and constructs a device to render into its window. This program renders its scene to the device for presentation in response to `WM_PAINT` or `WM_SIZING` messages.

An application needn't restrict itself to rendering in response to window damage. The SDK sample framework renders continuously by adjusting its message loop to render the scene whenever there are no messages to process, in addition to `WM_PAINT`.

The Direct3D specific parts of the program are those method calls on the `IDirect3D9` and `IDirect3DDevice9` interfaces, which are obtained in lines 133–164. The `frame_draw` procedure handles rendering of the scene in lines 51–61. The scene data, consisting of the definition for a single screen-space triangle, is given in lines 36–49.

Once the scene has been rendered, its ready for presentation on the monitor. Direct3D devices can become **lost**.The minimal application handles lost devices in lines 63–78 by waiting until the device can be regained. Device loss is discussed in the chapter 3.

For simplicity, the only error checking performed by `frame_draw` is the response to `Present` to check for a lost device. Most methods return a `HRESULT` which should always be checked as an aid to debugging during development. This and other debugging techniques are discussed in chapter 22.

Listing 2.1: `minimal.cpp`: A minimal well-behaved Direct3D application.

```
1   // minimal.cpp - minimal D3D application
2   //
3   // Originally by Sameer Nene of Microsoft, used with permission.
4   // Modified by Richard Thomson
5   //
6   #include <windows.h>
7   #include <tchar.h>
8   #include <atlbase.h>
9   #include <d3d9.h>
10  #include "rt/hr.h"
11
12  ////////////////////////////////////////////////////////////////////
13  // Simple HWND handling
14  class c_window
15  {
16  public:
17      typedef LRESULT CALLBACK
18          t_window_proc(HWND window, UINT msg, WPARAM wp, LPARAM lp);
19
20      c_window() : m_window(NULL)
21      {}
22      ~c_window();
23
24      void create(LPCTSTR name, LPCTSTR window_class,
25                  UINT width, UINT height,
26                  HINSTANCE instance, t_window_proc *proc);
27
28      operator HWND() const { return m_window; }
29
30  private:
31      HWND m_window;
32  };
33
34  void
35  c_window::create(LPCTSTR name, LPCTSTR window_class,
36                   UINT width, UINT height,
37                   HINSTANCE instance,
38                   c_window::t_window_proc *proc)
39  {
40      // Register the window class for the main window.
41      const WNDCLASS wc =
42      {
43          0, proc, 0, 0, instance, NULL, TWS(::LoadCursor(NULL, IDC_CROSS)),
44          static_cast<HBRUSH>(TWS(::GetStockObject(BLACK_BRUSH))),
```

```
45          NULL, window_class
46      };
47      TWS(::RegisterClass(&wc));
48
49      // Create the main window.
50      m_window = TWS(::CreateWindow(window_class, name,
51          WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
52          width, height, TWS(::GetDesktopWindow()), NULL,
53          instance, NULL));
54  }
55
56  c_window::~c_window()
57  {
58          // ::DefWindowProc calls ::DestroyWindow on WM_CLOSE
59  }
60
61  ////////////////////////////////////////////////////////////////////
62  // Simple application class
63  class c_d3d_application
64  {
65  public:
66      c_d3d_application();
67      ~c_d3d_application();
68
69      void create(HINSTANCE instance);
70
71      HWND window() const { return m_window; }
72
73  private:
74      void frame_draw();
75      LRESULT on_message(HWND window, UINT msg, WPARAM wp, LPARAM lp);
76
77      // device window and interface
78      c_window m_window;
79      CComPtr<IDirect3DDevice9> m_device;
80      // device presentation parameters
81      D3DPRESENT_PARAMETERS m_presentation;
82
83      // width and height of frame buffer
84      static const UINT FRAME_WIDTH = 320;
85      static const UINT FRAME_HEIGHT = 240;
86
87      static LRESULT CALLBACK
88          s_on_message(HWND window, UINT msg, WPARAM wp, LPARAM lp);
89      static c_d3d_application *s_app;
90  };
```

```
91
92  c_d3d_application *c_d3d_application::s_app = NULL;
93
94  c_d3d_application::c_d3d_application() :
95      m_window(),
96      m_device()
97  {
98      // must set this before doing anything that could process messages
99      s_app = this;
100 }
101
102 void
103 c_d3d_application::create(HINSTANCE instance)
104 {
105     m_window.create(_T("Minimal Direct3D9 Application"), _T("rt_Minimal"),
106             FRAME_WIDTH, FRAME_HEIGHT, instance, &s_on_message);
107
108     // Get IDirect3D9 interface
109     CComPtr<IDirect3D9> d3d;
110     d3d.Attach(::Direct3DCreate9(D3D_SDK_VERSION));
111     if (!d3d)
112     {
113         THR(E_NOINTERFACE);
114     }
115
116     // find out display mode format to use for back buffer
117     D3DDISPLAYMODE dm;
118     THR(d3d->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &dm));
119
120     D3DPRESENT_PARAMETERS presentation =
121     {
122         // width, height, format, number of back buffers
123         FRAME_WIDTH, FRAME_HEIGHT, dm.Format, 1,
124         // multisample type, swap effect, device window, windowed
125         D3DMULTISAMPLE_NONE, 0, D3DSWAPEFFECT_FLIP, m_window, TRUE,
126         // auto depth/stencil surface flag, format
127         FALSE, D3DFMT_UNKNOWN
128     };
129     m_presentation = presentation;
130
131     // now create the device
132     THR(d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
133         m_window, D3DCREATE_SOFTWARE_VERTEXPROCESSING,
134         &m_presentation, &m_device));
135
136     // display the window
```

```
137       ::ShowWindow(m_window, SW_SHOWDEFAULT);
138       TWS(::UpdateWindow(m_window));
139   }
140
141   c_d3d_application::~c_d3d_application()
142   {
143   }
144
145   // draw a frame consisting of a single
146   // white triangle on a black background
147   void
148   c_d3d_application::frame_draw()
149   {
150       if (!m_device)
151       {
152           return;
153       }
154
155       // structure for triangle vertex positions
156   #define VTX(x_, y_) \
157       { x_*FRAME_WIDTH, y_*FRAME_HEIGHT, 0.5f, 2.0f }
158       const struct MYVERTEX
159       {
160           float m_pos[4];                 // x, y, z, 1/w
161       }
162       triangle[] =
163       {
164           VTX(0.50f, 0.25f),
165           VTX(0.75f, 0.75f),
166           VTX(0.25f, 0.75f)
167       };
168   #undef VTX
169
170       // start the scene and clear the frame buffer
171       THR(m_device->BeginScene());
172       THR(m_device->Clear(0, NULL, D3DCLEAR_TARGET, 0, 1.0f, 0));
173
174       // describe triange vertex data to the device
175       THR(m_device->SetFVF(D3DFVF_XYZRHW));
176       THR(m_device->DrawPrimitiveUP(D3DPT_TRIANGLELIST, 1,
177                                     triangle, sizeof(MYVERTEX)));
178
179       // end 3D scene
180       THR(m_device->EndScene());
181
182       // present the scene & handle lost devices
```

```
183         HRESULT hr = m_device->Present(NULL, NULL, NULL, NULL);
184         while (D3DERR_DEVICELOST == hr)
185         {
186             do
187             {
188                 ::Sleep(1000);
189                 hr = m_device->TestCooperativeLevel();
190             }
191             while (hr != D3DERR_DEVICENOTRESET);
192
193             if (FAILED(m_device->Reset(&m_presentation)))
194             {
195                 hr = D3DERR_DEVICELOST;
196             }
197         }
198
199         // clear Win32 dirty region
200         TWS(::ValidateRect(m_window, NULL));
201     }
202
203     // basic windows message procedure
204     LRESULT CALLBACK
205     c_d3d_application::s_on_message(HWND window, UINT msg, WPARAM wp, LPARAM lp)
206     {
207             ATLASSERT(s_app != NULL);
208         return s_app->on_message(window, msg, wp, lp);
209     }
210
211     LRESULT
212     c_d3d_application::on_message(HWND window, UINT msg, WPARAM wp, LPARAM lp)
213     {
214         LRESULT result = 0;
215
216         if (WM_SIZING == msg || WM_PAINT == msg)
217         {
218             frame_draw();
219         }
220         else
221         {
222             result = ::DefWindowProc(window, msg, wp, lp);
223         }
224         return result;
225     }
226
227     ////////////////////////////////////////////////////////////////////////
228     int __stdcall
```

```
229  _tWinMain(HINSTANCE instance, HINSTANCE, LPTSTR, int)
230  {
231      try
232      {
233          c_d3d_application app;
234          app.create(instance);
235
236          // pump messages
237          MSG msg;
238          while (0 < ::GetMessage(&msg, app.window(), 0, 0))
239          {
240              ::TranslateMessage(&msg);
241              ::DispatchMessage(&msg);
242          }
243      }
244      catch (rt::hr_message &bang)
245      {
246          return rt::display_error(bang);
247      }
248      catch (...)
249      {
250          return -1;
251      }
252
253      return 0;
254  }
```

2.3 Direct3D Architecture

Direct3D is a Windows subsystem component at a level comparable to GDI, see figure 2.1. Direct3D differs from GDI in that it does not attempt anything more than presenting a thin abstraction of the graphics hardware. The Direct3D device driver deals directly with the graphics hardware. Direct3D communicates directly with the display driver and does not require GDI, achieving higher performance than possible with GDI rendering.

The main abstractions of Direct3D are devices, swap chains and resources. The **device** is the object that exposes the rendering operations of the hardware. The device's properties control the rendering behavior or provide information about rendering, while the device's methods are used to perform the rendering itself. Devices always contain at least one **swap chain** and a collection of **resources** used for rendering as shown in figure 2.2. Resources are application specific data stored in or near the device hardware for use during rendering. Direct3D provides resources for scene geometry (vertices and indices) and appearance (images, textures, and volumes).

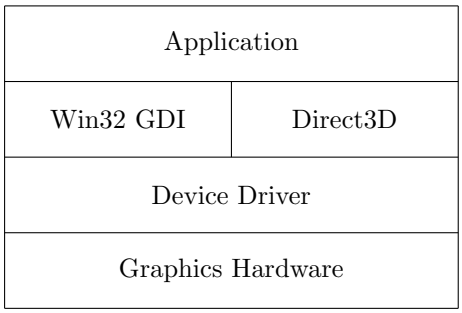

Figure 2.1: Relationship of Direct3D to other Windows subsystems.

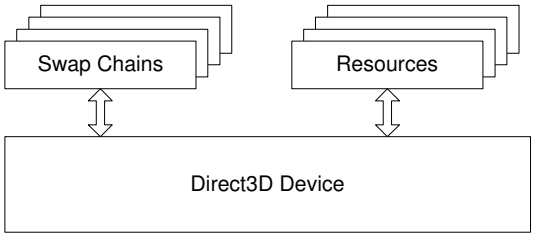

Figure 2.2: Direct3D device architecture

A **surface** is a resource containing a rectangular collection of pixel data such as color, alpha, depth/stencil or texture information. A swap chain contains one or more back buffer surfaces where scenes are rendered and presented for display on the monitor. A device's **render target** is the back buffer surface, with an optional depth/stencil surface, in which rendering will occur.

While all back buffers are valid render targets, not all render targets are back buffers. It is also possible to have a texture as a render target allowing dynamic rendering effects.

To obtain a device object, Direct3D provides an object for device enumeration and creation. All other objects are created through the device. An application first obtains the runtime interface, then selects and creates a device from those available, and using the device creates the necessary resources for rendering.

2.4 Direct3D HRESULTs

As mentioned in chapter 1, most COM methods return `HRESULT` to indicate success or failure. The SDK documentation lists common error codes returned by Direct3D interface methods, but the list is not meant to be exhaustive. The error codes listed are the ones that arise in typical use of the interface in question and in some senses can be considered "expected" errors, such as `D3DERR_OUT-`

OFVIDEOMEMORY which indicates resource exhaustion, not a programming error.

During development errors in method parameters to Direct3D objects will cause D3DERR_INVALIDCALL to be returned, indicating invalid parameters were passed to the method. Distinct error codes for all the possible argument failures are not provided, only a single code indicating this type of failure. Always check the return values from Direct3D interface methods when they return HRESULT. Many times, a bug in your application can be found sooner by isolating the first routine that returns D3DERR_INVALIDCALL. This error code is not considered an "expected" error, because it arises from improper use of the interface. These errors should be eliminated from your application if they occur. The debug runtime is an invaluable aid in determining the cause of D3DERR_INVALIDCALL and is discussed in chapter 22.

2.5 Windowed and Exclusive Mode

Direct3D devices have two distinct modes of operation: windowed mode and exclusive mode. In windowed mode, the results of graphic rendering are presented for display inside the client area of a window on the desktop. Direct3D cooperates with GDI to make the results of rendering visible, using a ::StretchBlt operation to present a back buffer in the window's client region.

In exclusive mode, Direct3D communicates directly with the display driver avoiding GDI. When an exclusive mode application is running, no other applications have access to the display hardware and no GDI output is visible on the screen. A request for exclusive mode may fail if the adapter is already under exclusive mode control by another application.

Supporting both windowed and exclusive mode is easy and should always be supported in an application. Windowed operation may be slower, but it is much easier to debug a Direct3D program in windowed mode than in exclusive mode. The selection between windowed and exclusive operation is done at the time a Direct3D device is created or reset.

2.6 Device Types

Each adapter can support devices of several types, operating in either exclusive or windowed mode. The three types of devices supported by Direct3D are the HAL device, the reference device and the pluggable software device. The device type is defined by the D3DDEVTYPE enumeration.

```
typedef enum _D3DDEVTYPE
{
    D3DDEVTYPE_HAL     = 1,
    D3DDEVTYPE_NULLREF = 4,
    D3DDEVTYPE_REF     = 2,
    D3DDEVTYPE_SW      = 3
} D3DDEVTYPE;
```

The HAL (hardware abstraction layer) device has hardware acceleration of graphics rendering, making it the fastest device type.

The reference device[1] is supplied only with the installation of the SDK. It includes a software implementation of the entire pipeline feature set. The implementation is tuned for accuracy and not for speed. This device will always be the slowest device available, but it is very useful for debugging. When developing software applications that target new features not yet common in hardware, the reference device may be your only choice for generating an image with those Direct3D features.

The "null reference device" does nothing and all rendering will result in a black screen. The null reference device will also be returned when the SDK is not installed on a machine and the application requests the reference device. You can detect the presence of the null reference device by examining the `D3DPMISCCAPS_NULLREFERENCE` bit of the `PrimitiveMiscCaps` member of `D3DCAPS9` on the device. The null reference device can be useful for manipulating resources on a machine that provides no hardware or software implementation of the runtime.

The pluggable software device is provided through the `RegisterSoftwareDevice` method. The interface for a software pluggable device is currently undocumented and no software pluggable rendering devices are available in DirectX 9.0c.[2] If software rendering is a requirement for your application, you should consider writing your own software renderer or using a previous version of the Direct3D interfaces that support software rendering. Writing a software renderer is beyond the scope of this book and the pluggable device is mentioned only here for completeness.

2.7 Resources

Collectively, the term "resources" refers to all the additional information beyond device properties needed to render a scene. Before we discuss how to create a device, we will briefly summarize the characteristics common to all resources.

Every resource has `Type`, `Pool`, `Format`, and `Usage` attributes. Each of these attributes of a resource are specified at the time the resource is created and remain constant for the lifetime of the resource object. The `Type` attribute describes the kind of resource and is defined by the `D3DRESOURCETYPE` enumeration.

```
typedef enum _D3DRESOURCETYPE {
    D3DRTYPE_SURFACE       =  1,
    D3DRTYPE_VOLUME        =  2,
    D3DRTYPE_TEXTURE       =  3,
    D3DRTYPE_VOLUMETEXTURE =  4,
```

[1] Also referred to as the reference rasterizer, or `REFRAST`

[2] A software rasterizer for DirectX 9.0c was later created; see "Software Rasterizer for the Microsoft DirectX 9.0 Software Development Kit" `http://www.microsoft.com/downloads/details.aspx?FamilyID=57d03491-6c49-4429-bff7-130408b5f410&DisplayLang=en`

```
    D3DRTYPE_CUBETEXTURE   =  5,
    D3DRTYPE_VERTEXBUFFER  =  6,
    D3DRTYPE_INDEXBUFFER   =  7
} D3DRESOURCETYPE;
```

The `Pool` attribute of a resource describes how it is managed by the Direct3D runtime and is defined by the `D3DPOOL` enumeration. Resources in the default pool exist only in device memory. Resources in the managed pool exist in system memory and will be copied into the device's memory by the runtime when needed. Resources in the system memory pool exist only in system memory. Resources in the scratch pool reside only in system memory and are not bound by format constraints of the device. When a device is lost, all resources in the default pool are lost and should be released and recreated by the application when the device is regained. Lost devices are discussed in detail in chapter 3.

```
typedef enum _D3DPOOL {
    D3DPOOL_DEFAULT   = 0,
    D3DPOOL_MANAGED   = 1,
    D3DPOOL_SYSTEMMEM = 2,
    D3DPOOL_SCRATCH   = 3
} D3DPOOL;
```

The `Format` attribute of a resource describes the layout of the resource's data in memory and is defined by the `D3DFORMAT` enumeration. All resources have a format, but most of the format enumerants define the memory layout for pixel data. The layout is indicated by the name of the enumerant, with the components of the pixel data listed from left to right in most significant bit first order. `D3DFMT_R8G8B8` indicates a 24-bit sized quantity with `R` in the most significant 8 bits, `G` in the middle 8 bits, and `B` in the least significant 8 bits. An `X` component indicates bits that are present in the pixel data but are unused. `D3DFMT_X8R8G8B8` indicates a 32-bit sized quantity with the most significant 8 bits unused and the remaining 24 bits allocated as in `D3DFMT_R8G8B8`.

```
typedef enum _D3DFORMAT
{
    D3DFMT_UNKNOWN       =  0,

    D3DFMT_INDEX16       =101,
    D3DFMT_INDEX32       =102,
    D3DFMT_VERTEXDATA    =100,

    D3DFMT_A4L4          = 52,
    D3DFMT_A8            = 28,
    D3DFMT_L8            = 50,
    D3DFMT_P8            = 41,
    D3DFMT_R3G3B2        = 27,
```

```
    D3DFMT_A1R5G5B5      = 25,
    D3DFMT_A4R4G4B4      = 26,
    D3DFMT_A8L8          = 51,
    D3DFMT_A8P8          = 40,
    D3DFMT_A8R3G3B2      = 29,
    D3DFMT_L16           = 81,
    D3DFMT_L6V5U5        = 61,
    D3DFMT_R16F          = 111,
    D3DFMT_R5G6B5        = 23,
    D3DFMT_V8U8          = 60,
    D3DFMT_X1R5G5B5      = 24,
    D3DFMT_X4R4G4B4      = 30,

    D3DFMT_R8G8B8        = 20,

    D3DFMT_A2R10G10B10   = 35,
    D3DFMT_A2B10G10R10   = 31,
    D3DFMT_A2W10V10U10   = 67,
    D3DFMT_A8B8G8R8      = 32,
    D3DFMT_A8R8G8B8      = 21,
    D3DFMT_CxV8U8        = 117,
    D3DFMT_G16R16        = 34,
    D3DFMT_G16R16F       = 112,
    D3DFMT_Q8W8V8U8      = 63,
    D3DFMT_R32F          = 114,
    D3DFMT_V16U16        = 64,
    D3DFMT_W11V11U10     = 65,
    D3DFMT_X8L8V8U8      = 62,
    D3DFMT_X8B8G8R8      = 33,
    D3DFMT_X8R8G8B8      = 22,

    D3DFMT_A16B16G16R16  = 36,
    D3DFMT_A16B16G16R16F = 113,
    D3DFMT_G32R32F       = 115,
    D3DFMT_Q16W16V16U16  =110,

    D3DFMT_A32B32G32R32F = 116,

    D3DFMT_DXT1          = MAKEFOURCC('D', 'X', 'T', '1'),
    D3DFMT_DXT2          = MAKEFOURCC('D', 'X', 'T', '2'),
    D3DFMT_DXT3          = MAKEFOURCC('D', 'X', 'T', '3'),
    D3DFMT_DXT4          = MAKEFOURCC('D', 'X', 'T', '4'),
    D3DFMT_DXT5          = MAKEFOURCC('D', 'X', 'T', '5'),
    D3DFMT_G8R8_G8B8     = MAKEFOURCC('G', 'R', 'G', 'B'),
    D3DFMT_R8G8_B8G8     = MAKEFOURCC('R', 'G', 'B', 'G'),
    D3DFMT_UYVY          = MAKEFOURCC('U', 'Y', 'V', 'Y'),
```

```
    D3DFMT_YUY2          = MAKEFOURCC('Y', 'U', 'Y', '2'),

    D3DFMT_MULTI2_ARGB8  = MAKEFOURCC('M','E','T','1'),

    D3DFMT_D15S1         = 73,
    D3DFMT_D16           = 80,
    D3DFMT_D16_LOCKABLE  = 70,
    D3DFMT_D32F_LOCKABLE = 82,

    D3DFMT_D24S8         = 75,
    D3DFMT_D24FS8        = 83,
    D3DFMT_D24X4S4       = 79,
    D3DFMT_D24X8         = 77,
    D3DFMT_D32           = 71
} D3DFORMAT;
```

The `An`, `Ln`, `Pn`, `Rn`, `Gn`, and `Bn` channels are encoded as unsigned quantities in the range $[0, 2^n - 1]$. The `Un`, `Vn`, `Wn`, `Qn` channels are encoded as signed quantities in the range $[-2^{n-1}, 2^{n-1} - 1]$. The `Dn` and `Sn` channels encode depth/stencil surface data in a device-specific manner.

Some of the enumerants have values defined by the `MAKEFOURCC` macro, or so-called "four character codes." Additional vendor-specific resource formats can be introduced by using new four character codes. The `YUVY` and `YUY2` formats are pixel formats originating in digital video[3] and are only mentioned here for completeness. The `DXTn` formats are compressed texture formats defined by DirectX and discussed in chapter 11.

`D3DCOLOR`'s pixel format is `D3DFMT_A8R8G8B8`, whereas the pixel format of a `PALETTEENTRY` or `COLORREF` has the red and blue components reversed as if they were the fictitious `D3DFMT_X8B8G8R8`. If you mix GDI colors with `D3DCOLOR`s, you will have to convert them for use with Direct3D. Direct3D does not use `COLORREF` in any of its arguments, although `PALETTEENTRY` is used with palette texture formats. When Direct3D uses a `PALETTEENTRY`, it interprets the `peFlags` member of the structure as an 8-bit alpha channel as if it were the fictitious `D3DFMT_A8B8G8R8`.

The `Usage` attribute describes how the application will be use the resource and is defined by a collection of flag bits. These flags allow the runtime – and consequently the driver – to know which application resources are used in a static or dynamic access pattern. Static resources are typically loaded with data once and used repeatedly without change, while dynamic resources are repeatedly modified by the application. While every resource has non-zero `Type`, `Pool` and `Format` attributes, the `Usage` attribute may often be zero. Not all of the usage flags are appropriate for all resource types. The relevant flags are noted where those resources are discussed.

```
#define D3DUSAGE_AUTOGENMIPMAP      0x00000400L
```

[3] See ⟨`http://www.webartz.com/fourcc/indexyuv.htm`⟩

```
#define D3DUSAGE_DEPTHSTENCIL       0x00000002L
#define D3DUSAGE_DMAP               0x00004000L
#define D3DUSAGE_DONOTCLIP          0x00000020L
#define D3DUSAGE_DYNAMIC            0x00000200L
#define D3DUSAGE_NPATCHES           0x00000100L
#define D3DUSAGE_POINTS             0x00000040L
#define D3DUSAGE_RENDERTARGET       0x00000001L
#define D3DUSAGE_RTPATCHES          0x00000080L
#define D3DUSAGE_SOFTWAREPROCESSING 0x00000010L
#define D3DUSAGE_WRITEONLY          0x00000008L
```

2.8 IDirect3D9

A Direct3D application starts by obtaining the `IDirect3D9` COM interface pointer by calling `::Direct3DCreate9`, the only global function in the Direct3D core interfaces.

```
#define D3D_SDK_VERSION 31

IDirect3D9 *WINAPI ::Direct3DCreate9(UINT sdk_version);
```

The `version` argument must be `D3D_SDK_VERSION`. When the Direct3D header files are changed, this number is incremented. If the wrong version number is passed to `::Direct3DCreate9`, then the function will fail and return `NULL`. This ensures that the application used the final release header files and not header files from interim releases of the SDK.

`IDirect3D9` exposes a model of graphics hardware where each monitor is connected to an adapter, identified by an unsigned positive number. An adapter is not necessarily equivalent to a display card; in recent years hardware densities have increased to support two adapters on a single display card, so-called "dual head" displays. `IDirect3D9` exposes each of these displays as distinct adapters, even though there is only one graphics card.

This interface provides a consistent way for applications to examine all the adapters connected to a system and select one that is most appropriate for its rendering task. An application can also render to multiple devices which is useful for splitting different visual elements of the application across multiple monitors. The `IDirect3D9` interface is summarized in interface 2.1.

Interface 2.1: Summary of the `IDirect3D9` interface.

IDirect3D9 : IUnknown

Read-Only Properties	
`GetAdapterCount`	The number of adapters in the system.
`GetAdapterDisplayMode`	Current video display mode for an adapter.

`GetAdapterIdentifier`	Identifying information about a device.
`GetAdapterModeCount`	The number of supported video display modes on an adapter.
`GetAdapterMonitor`	The `HMONITOR` for an adapter.
`GetDeviceCaps`	The generic capabilities of a device.

Methods

`CheckDepthStencilMatch`	Check a device for a supported depth stencil surface for use with a render target and adapter display mode format.
`CheckDeviceFormat`	Check a device for supported resource types and formats.
`CheckDeviceFormatConversion`	TODO
`CheckDeviceMultiSampleType`	Check multisampling support for a device.
`CheckDeviceType`	Check an adapter for device type and windowed or exclusive mode support.
`CreateDevice`	Create a device on an adapter.
`EnumAdapterModes`	Obtain the supported display mode information for an adapter.
`RegisterSoftwareDevice`	Registers a software device with Direct3D.

```
interface IDirect3D9 : IUnknown
{
 //-------------------------------------------------------------
 // read-only properties
     UINT GetAdapterCount();
  HRESULT GetAdapterDisplayMode(UINT adapter,
              D3DDISPLAYMODE *value);
  HRESULT GetAdapterIdentifier(UINT adapter,
              DWORD flags,
              D3DADAPTER_IDENTIFIER9 *value);
     UINT GetAdapterModeCount(UINT adapter,
              D3DFORMAT format);
 HMONITOR GetAdapterMonitor(UINT adapter);
  HRESULT GetDeviceCaps(UINT adapter,
              D3DDEVTYPE device_kind,
              D3DCAPS9 *value);

 //-------------------------------------------------------------
 // methods
  HRESULT CheckDepthStencilMatch(UINT Adapter,
              D3DDEVTYPE device_kind,
```

```
            D3DFORMAT adapter_fmt,
            D3DFORMAT render_target_fmt,
            D3DFORMAT depth_stencil_fmt);
  HRESULT CheckDeviceFormat(UINT adapter,
            D3DDEVTYPE device_kind,
            D3DFORMAT adapter_fmt,
            DWORD usage,
            D3DRESOURCETYPE resource_kind,
            D3DFORMAT surface_fmt);
  HRESULT CheckDeviceFormatConversion(UINT adapter,
            D3DDEVTYPE device_kind,
            D3DFORMAT source_fmt,
            D3DFORMAT target_fmt);
  HRESULT CheckDeviceMultiSampleType(UINT adapter,
            D3DDEVTYPE device_kind,
            D3DFORMAT target,
            BOOL windowed,
            D3DMULTISAMPLE_TYPE sampling,
            DWORD *quality_levels);
  HRESULT CheckDeviceType(UINT adapter,
            D3DDEVTYPE device_kind,
            D3DFORMAT display_fmt,
            D3DFORMAT back_fmt,
            BOOL windowed);
  HRESULT CreateDevice(UINT adapter,
            D3DDEVTYPE device_kind,
            HWND focus_window,
            DWORD behavior_flags,
            D3DPRESENT_PARAMETERS *presentation,
            IDirect3DDevice9 **result);
  HRESULT EnumAdapaterModes(UINT adapter,
            D3DFORMAT format,
            UINT mode,
            D3DDISPLAYMODE *value);
  HRESULT RegisterSoftwareDevice(void *init_func);
};
```

2.9 Selecting a Device

Typically an application will enumerate all devices in the system and pick the device most suitable for the application's needs. First, call `GetAdapterCount` to determine the number of adapters in the system. Each adapter provides a number of video display modes. Each display mode contains a screen dimension, refresh rate and pixel format and is described by a `D3DDISPLAYMODE` structure:

```
typedef struct _D3DDISPLAYMODE
```

```
{
    UINT      Width;
    UINT      Height;
    UINT      RefreshRate;
    D3DFORMAT Format;
} D3DDISPLAYMODE;
```

The `Format` member will be either an RGB or XRGB format. The back buffer surface format of a device must be compatible wih the display mode's format. The `CheckDeviceFormat` method can be used to discover compatible formats. Typically, the back buffer surface format is the same pixel depth and color layout as the display format. An XRGB display format can be used with an ARGB back buffer of the same depth, provided that the corresponding ARGB format is validated by `IDirect3D9`.

The number of display modes supported by the adapter is returned by `GetAdapterModeCount` and the display mode information is returned by `EnumAdapterModes`. The display mode currently in use by the adapter is returned by `GetAdapterDisplayMode`.

Two display modes can have the same `Width`, `Height` and `Format` values but different values for the `RefreshRate`. When examining supported display modes, you may wish to ignore display modes that differ only in their refresh rates. In this situation, the application should prefer the display mode with a refresh rate corresponding to the current display mode to avoid a change in the monitor's refresh rate.

With a list of supported display formats, we are now ready to query for supported device types. The `CheckDeviceType` method tells us if a particular combination of display format and back buffer format are valid for a device of a given type operating in windowed or exclusive mode.

With a valid device type, we can check the device's capabilities for rendering features required by our application with `GetDeviceCaps`. Device capabilities are defined by the `D3DCAPS9` structure, described in chapter 3.

Next, we can validate all the resources required by our application with the `CheckDeviceFormat`. `CheckDeviceFormat` should be used to validate all the format of all resources used by the application: back buffer surfaces, depth/stencil surfaces, texture surfaces, and volume texture formats. Vertex buffer resources only have one format and are always valid. `D3DFMT_INDEX16` is always supported for index buffer and `D3DFMT_INDEX32` is valid for index buffers if the `MaxVertexIndex` member of `D3DCAPS9` is greater than $2^{16} - 1$.

Next, if the application requires a depth buffer for visibility determination, it should use `CheckDepthStencilMatch` to find a depth buffer that can be used with its render target formats in a given display mode. This check accommodates hardware that has additional requirements for a depth/stencil buffer that vary with the format of the render target and display mode. For example, some nVidia hardware requires that the depth buffer match the render target in depth.

Finally, an application with multisampling needs can validate its requirements with the `CheckDeviceMultiSampleType` method. Applications not concerned with multisampling can skip that check. For more about multisampling see chapter 14.

All of the check methods return `S_OK` if the requested combination is supported and `D3DERR_NOTAVAILABLE` otherwise.

2.10 Determining Available Resource Memory

Having found a device capable of supporting the application's needs, it may be desirable to determine if the device has sufficient memory resources for the application. If an application has specific memory requirements, the only way to determine if a device passes those requirements is to instantiate the device and attempt to create the necessary resources.

The amount of resource memory available to an application depends on the display mode in use. Creating an exclusive mode device can cause jarring display mode changes if the device's requested video mode differs from the video mode used by the desktop. To avoid this unnerving effect every time the application starts, an application can perform the memory test once at setup time and cache the results. This is safe to do because the amount of memory available on a device in a particular display mode will not change unless the hardware is changed. When the hardware is changed, the user can run the application setup again to cache the new hardware information.

2.11 Device Capabilities

Graphics devices have a wide variety of architectures, hardware capabilities and performance ranges. A demanding application will want to probe the hardware's capabilities to determine an appropriate rendering strategy most consistent with the application's goals. The application can also use hardware capabilities to reject devices inappropriate for its rendering strategy.

An application calls `GetDeviceCaps` to obtain the generic capabilities of a particular type of device on an adapter. The device capabilities are described by the `D3DCAPS9` structure. An overview of this structure is given in section 3.3. The capabilities structure covers many aspects of the device. Specific device capabilities are discussed with the relevant portion of the graphics pipeline and miscellaneous device capabilities are discussed in chapter 3.

2.12 Identifying a Particular Device

In a perfect world, we could weed out incapable devices with the above checks for its support and capabilities. Unfortunately, we do not live in a perfect world, we live in a world of imperfect beings that write software – including device drivers! This ugly reality brings us to `GetAdapterIdentifier` which allows

an application to identify a specific brand of adapter from a specific vendor. Sometimes the only way to deal with the "conditions in the field" is to identify cards using this method, although it should only be used as a last resort to identify particular problem situations for your application.

`GetAdapterIdentifier` returns a `D3DADAPTER_IDENTIFIER9` structure identifying this adapter in the system:

```
typedef struct _D3DADAPTER_IDENTIFIER9
{
  char           Driver[MAX_DEVICE_IDENTIFIER_STRING];
  char           Description[MAX_DEVICE_IDENTIFIER_STRING];
  char           DeviceName[32];

#ifdef _WIN32
  LARGE_INTEGER DriverVersion;
#else
  DWORD          DriverVersionLowPart;
  DWORD          DriverVersionHighPart;
#endif

  DWORD          VendorId;
  DWORD          DeviceId;
  DWORD          SubSysId;
  DWORD          Revision;
  GUID           DeviceIdentifier;
  DWORD          WHQLLevel;
  HMONITOR       hMonitor;
} D3DADAPTER_IDENTIFIER9;
```

The `Driver` and `Description` fields provide human readable text intended to be used to guide a user in selecting this device from a user interface. The `DriverVersion` field indicates the version of the Direct3D driver. Comparison operations are valid on the entire 64 bit quantity. The driver version can be further subdivided into `Product`, `Version`, `SubVersion` and `Build` fields with the `HIWORD` and `LOWORD` Win32 macros. `HIWORD` and `LOWORD` take a 32-bit quantity and return the most significant 16 bits and the least significant 16 bits, respectively.

Member	HIWORD	LOWORD
DriverVersion.LowPart	Product	Version
DriverVersion.HighPart	SubVersion	Build

The `VendorId`, `DeviceId`, `SubSysId` and `Revision` fields are used to differentiate among different hardware chipsets. The values may be zero if the chipset information is unknown. The `DeviceIdentifier` field contains a GUID that uniquely identifies this driver/chipset pair. The GUID should be used to identify changes in graphics hardware configurations and to track problematic drivers,

if necessary. The `hMonitor` field gives the handle of the monitor connected to this adapter.

The `WHQLLevel` field gives the **WHQL** certification information for this driver. The value is either zero indicating that the driver is not certified, one indicating that the driver is certified but no date information is known, or a packed date structure. The packed date indicates the latest release of the WHQL certification test passed by this driver. The date is packed as follows:

WHQL: Windows Hardware Quality Laboratory

Bits	Description
0-7	Day, $[1, 31]$
8-15	Month, $[1, 12]$
16-31	Year, $[1999, 65535]$

Determining the WHQL level can be a costly operation. This operation can be avoided by specifying 0 for the `flag` argument to `GetAdapterIdentifier`. If the WHQL information is desired, pass `D3DENUM_WHQL_LEVEL` for `flags`. Obtaining WHQL information may cause the runtime to download new WHQL certificates from the internet.

```
#define D3DENUM_WHQL_LEVEL    0x00000002L
```

2.13 Creating the Device

After a device has been selected from those provided by the system, the application creates an instance of the `IDirect3DDevice9` interface with the `CreateDevice` method. This method returns a device interface that is ready for rendering.

The `focus_window` argument designates the window that will be the focus for this device. For a device operating in exclusive mode, this window must be a top-level window. Applications should not create a device in response to `WM_CREATE`. Direct3D applications will receive messages on the focus window when the device is created and in exclusive mode, Direct3D will subclass the focus window.[4] An MFC application should obtain the focus window handle from the `::AfxGetSafeWindow` function.

The `behavior_flags` argument specifies global behaviors of the created device and is defined by a collection of the following flags.

```
#define D3DCREATE_ADAPTERGROUP_DEVICE           0x00000200L
#define D3DCREATE_DISABLE_DRIVER_MANAGEMENT     0x00000100L
#define D3DCREATE_DISABLE_DRIVER_MANAGEMENT_EX  0x00000400L
#define D3DCREATE_FPU_PRESERVE                  0x00000002L
#define D3DCREATE_HARDWARE_VERTEXPROCESSING     0x00000040L
#define D3DCREATE_MIXED_VERTEXPROCESSING        0x00000080L
#define D3DCREATE_MULTITHREADED                 0x00000004L
#define D3DCREATE_NOWINDOWCHANGES               0x00000800L
```

[4] "Subclassing" here refers to Win32 window subclassing, not C++ subclassing.

```
#define D3DCREATE_PUREDEVICE                    0x00000010L
#define D3DCREATE_SOFTWARE_VERTEXPROCESSING     0x00000020L
```

Some display cards provide multiple video outputs on a single card. `D3DCREATE_ADAPTERGROUP_DEVICE` allows an application to drive both video outputs through a single device interface, allowing resources to be shared for both outputs. See section 2.15 for more about using `CreateDevice` with this flag.

The `D3DCREATE_DISABLE_DRIVER_MANAGEMENT` and `D3DCREATE_DISABLE_DRIVER_MANAGEMENT_EX` flags disable management of device resources by the driver and force all resource management to occur in the runtime. The ex form of the flag returns errors from resource creation methods to the application upon insufficient device memory, while the non-ex form manages device memory through the runtime and does not expose insufficient video memory to the application.

Direct3D uses single precision floating-point computations. If an application requires higher precision from the FPU, there are two choices. Either the application can ensure that the FPU is in single precision mode when calling into Direct3D, or it can request that the device preserve the application's FPU precision before Direct3D performs any floating-point operations and restore the precision before returning to the application.

Direct3D also has some thread sensitivities that must be observed in multithreaded applications. The multithreaded create flag instructs Direct3D that multiple threads will use the device. In this situation, Direct3D will use a global critical section to ensure serialized access to the device. If a multithreaded application can safely ensure that no two threads will use a device simultaneously, then this flag can be omitted. It is recommended that multithreaded applications use the flag until a performance measurement indicates the overhead of the critical section to be significant.

The pure device flag allows an application to create a device with the minimal amount of internal state tracking. As we will see in the next chapter, devices have a large amount of state that controls their operation. With the pure device flag, the runtime does a minimal amount of checking on this state and passes it directly to the driver. This can provide a small performance boost for an application, at the cost of some ease of use. Pure devices will be discussed in more detail in chapter 3.

The vertex processing flags allow the application to select pure software vertex processing, pure hardware vertex processing or a mix of software and hardware vertex processing. `D3DCREATE_SOFTWARE_VERTEXPROCESSING` selects software vertex processing, which is always available from the runtime. The runtime uses an efficient implementation of software vertex processing that is optimized for the CPU. `D3DCREATE_MIXED_VERTEXPROCESSING` selects a combination of software and hardware vertex processing selected by `SetSoftwareVertexProcessing`. Mixed vertex processing is incompatible with a pure device and will fail if both are requested together. A device provides hardware vertex processing when the `D3DDEVCAPS_HWTRANSFORMANDLIGHT` bit of `D3DCAPS9::DevCaps` is set. Vertex processing is described in detail in section 6.2.

The `presentation` argument describes how the device will present new

renderings for display on the monitor with values stored in the D3DPRESENT_-PARAMETERS structure. The presentation parameters describe the behavior of presentation for the device under windowed and full-screen modes. Some of the values in this structure may be updated by the runtime before returning from this function, so the parameter should point to a writable region of memory.

```
typedef struct _D3DPRESENT_PARAMETERS_
{
    UINT                BackBufferWidth;
    UINT                BackBufferHeight;
    D3DFORMAT           BackBufferFormat;
    UINT                BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD               MultiSampleQuality;
    D3DSWAPEFFECT       SwapEffect;
    HWND                hDeviceWindow;
    BOOL                Windowed;
    BOOL                EnableAutoDepthStencil;
    D3DFORMAT           AutoDepthStencilFormat;
    DWORD               Flags;
    UINT                FullScreen_RefreshRateInHz;
    UINT                PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

The BackBufferWidth, BackBufferHeight, BackBufferFormat, BackBuffer-Count, MultiSampleType, MultiSampleQuality and SwapEffect members describe the swap chain created with the device. The details of a swap chain and presentation are discussed in chapter 4.

The Windowed member differentiates between windowed operation and exclusive operation. All DirectX 9.0c devices support windowed operation.

The AutoDepthStencil and AutoDepthStencilFormat members control the automatic allocation of a depth/stencil buffer when the device is created. When enabled, the format member describing the depth/stencil surface must be a format validated by the adapter with CheckDepthStencilMatch. Allocating depth/stencil buffer is described in detail in chapter 5.

The Flags member can be zero or one or more of the following values:

```
#define D3DPRESENTFLAG_DEVICECLIP           0x00000004
#define D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL 0x00000002
#define D3DPRESENTFLAG_LOCKABLE_BACKBUFFER  0x00000001
#define D3DPRESENTFLAG_VIDEO                0x00000010
```

The D3DPRESENTFLAG_DEVICECLIP flag restricts the results of a Present operation to the client area of the device window in windowed mode. This flag is supported only on Windows 2000 and Windows XP. The D3DPRESENTFLAG_-VIDEO flag is a hint to the device that the back buffer contains video data.

The D3DPRESENTFLAG_DISCARDDEPTHSTENCIL flag instructs the runtime to discard the contents of the depth stencil surface after a call to Present, or when a new depth stencil surface is set on the device. This effectively makes the depth stencil surface a write only surface, allowing the device to improve rendering performance. An error occurs if you try to set this flag and request one of the lockable depth stencil surface formats D3DFMT_D16_LOCKABLE or D3DFMT_D32F_LOCKABLE.

The D3DPRESENTFLAG_LOCKABLEBACKBUFFER flag requests a default swap chain with back buffer surfaces that can be directly accessed by the application. Accessing surface data directly with the IDirect3DSurface9 interface is described in chapter 4.

The remaining members of the presentation parameters have differing interpretations under windowed and full-screen modes.

In windowed mode, hDeviceWindow specifies the window whose client area will be used for presentation in windowed operation. If hDeviceWindow is zero, then the focus window will be used for presentation. The FullScreen_RefreshRateInHz member must be zero for windowed operation. The presentation intervals supported by windowed mode are listed in table 4.1.

In exclusive mode, hDeviceWindow specifies the top-level window used by the application. If multiple devices are created in exclusive mode, only one device can use the focus_window for hDeviceWindow, with the remaining devices using their own top-level windows for hDeviceWindow. The BackBufferWidth, BackBufferHeight, and BackBufferFormat members must be equal to the respective members of a valid D3DDISPLAYMODE for this adapter. The FullScreen_PresentationInterval specifies the desired relationship between the presentation rate and the screen refresh rate. It is described in section 4.6. The FullScreen_RefreshRateInHz can be a refresh rate of a valid D3DDISPLAYMODE consistent with the rest of the members, or the following special values.

```
#define D3DPRESENT_RATE_DEFAULT   0x00000000
```

D3DPRESENT_RATE_DEFAULT instructs the runtime to choose a suitable refresh rate in exclusive mode, and uses the current refresh rate in windowed mode.

When a device is created in exclusive mode, the adapter is placed into the requested display mode. Once a device has been created, the application will construct any resources it needs for rendering and then present a sequence of renderings to the monitor for display.

2.14 Multiple Monitors

For systems with multiple monitors, the virtual desktop consists of a bounding rectangle containing all the adapters in the system that participate in the Windows desktop. Secondary adapters that do not participate in the desktop may also be attached to the system. Secondary adapters may be enumerated by IDirect3D9 if support is available for the adapter. All adapters on the desktop share at least a 1 pixel boundary in common with other monitors on the

desktop. When the cursor is moved off screen at a boundary pixel adjacent to another adapter's display, the cursor moves to that adapter's display. For more information about multiple monitors on Windows operating systems see the MSDN documentation.

An application may wish to create a full-screen display on a specific monitor. The `GetAdapterMonitor` method returns an `HMONITOR` handle for an adapter. Once you have the handle to the device's monitor you can determine what part of the virtual desktop is covered by the monitor.

For example, a driving simulation written in Direct3D could create a single simulated image across a row of monitors arranged horizontally. using the GDI function `::GetMonitorInfo` you can determine the portion of the virtual desktop described by each `HMONITOR`. Using the `HMONITOR` for an adapter you can then partition the rendering of the scene across the devices. The screen saver framework in the SDK provides an example of using multiple monitors in a Direct3D application.

If your application is running in windowed mode, the DirectX runtime handles the partitioning of your scene when your device window spans multiple monitors on the desktop. When your application presents its rendering for display, the runtime copies the appropriate pixel data to each device. An application can also create devices on the appropriate adapters and partition the scene itself. For full-screen multimonitor applications, devices must be created on each monitor separately.

2.15 Adapter Group Devices

Multihead adapters provide distinct video outputs from a single card. When a device is created with `D3DCREATE_ADAPTERGROUP_DEVICE`, instead of providing a single `D3DPRESENT_PARAMETERS` structure, an array of `D3DPRESENT_PARAMETERS` structures are supplied instead. The number of structures in the array is at least as large as the `NumberOfAdaptersInGroup` member of `D3DCAPS9`. No matter how many swap chains are created on an adapter group device, only a single depth stencil surface will be created if requested.

Only one of the presentation parameter structures in the array passed to `CreateDevice` can use the focus window parameter as its device window. The remaining structures must use their own distinct top-level windows as the device window.

An adapter group device has a swap chain associated with each monitor in the adapter group. In this way, each monitor can share rendering resources between them and the presentation of all monitors in the adapter group is performed with a single call to `Present`.