Chapter 3

Direct3D Devices

"Lay down a method for everything, and stick to it inviolably, as far as unexpected incidents may allow."

Lord Chesterfield: *Letter to his son*, February 5, 1750

3.1 Overview

This chapter introduces the Direct3D device object which provides an abstraction of the rendering pipeline. The operation of the pipeline is configured through the device's property methods. Images are drawn on the device through 2D pixel copy operations, or through the rendering of three dimensional scenes.

A scene is a collection of three dimensional objects that are projected onto the render target surface from the viewpoint of a synthetic camera. Each object in a scene is described by a collection of geometric primitives, such as points, lines and triangles, with the device's action methods.

The pipeline converts geometric descriptions into pixels on the render target surface through the process of **rasterization**. Graphic primitives are rendered in the order in which they are described to the devices similar to the way a CPU executes binary instructions sequentially through memory.

The entire graphics pipeline is controlled through the properties of the device. The properties are manipulated through the `Get` and `Set` methods of the device. In addition to the fixed-function pipeline provided by traditional graphics APIs, Direct3D provides a programmable graphics pipeline through the use of vertex and pixel **shaders**.

Every device has a distinct set of **capabilities**. Direct3D specifies an abstract machine interface, but does not provide a software emulation for a feature not provided directly by the hardware's driver. A device provides specific information on its capabilities to allow an application to adapt to the capabilities of the device.

Devices are containers for **resource** objects. The resource management provided by Direct3D is presented and the `IDirect3DResource9` interface is summarized.

Much of the behavior of the graphics pipeline is controlled by **render states**. There are a large number of render states which are summarized in this chapter. Each section of the pipeline will discuss the specifics of its associated render states, if any.

An application frequently needs to set groups of render states. A code idiom is presented for making this easier. Groups of device properties, including render states, can be cached and set as a group with **state blocks**.

3.2 `IDirect3DDevice9`

The `IDirect3DDevice9` interface controls the behavior of the pipeline and contains a large number of properties and methods. In the subsequent chapters we will explore each property of the device and how it affects rendering. The device interface is summarized in interface 3.1. The device method prototypes are given in the text when they are introduced.

The device object represents the rendering pipeline. We get images from the pipeline by supplying it with scene data and instructing it to render scenes into images. Scene data consists of geometric data defining shapes in space and data defining the appearance of the shapes.

We can break the pipeline up into large sections consisting of vertex data assembly, vertex processing, primitive assembly and rasterization, pixel processing, the frame buffer and video scan out. The vertex data assembly section gathers together vertex components from a collection of data streams to assemble a complete vertex and its associated data. Vertex processing performs computations on each vertex such as the transformation and lighting of vertices. The processed vertex data is then assembled into graphic primitives and rasterized into a stream of pixels. Pixel processing performs computations on each rasterized pixel to determine the final color of the pixel that will be written into the frame buffer. The frame buffer performs a read/modify/write operation combining processed pixels from the rasterizer with the existing pixels in the render target. Video scan out reads the pixels out of the frame buffer for conversion into video signals displayed by the monitor.

Table 3.1 organizes the methods on the device interface by pipeline section, table 3.2 organizes the members and flags in the capabilities structure by pipeline section, and table 3.5 organizes the render states and texture stage states by pipeline section. The poster accompanying this book expands on the figures presented in this chapter with a detailed depiction of the entire graphics pipeline, their associated methods, device capabilities and render states.

Scenes are rendered to the render target selected on the device. If the render target is part of a swap chain, the render target can be made visible on the device through presentation. (The render target may not be part of a swap chain if the render target is a texture resource.) You present renderings for display through

a swap chain as the last thing you do when rendering a scene.

A swap chain consists of one or more back color buffers into which images are rendered. A device is always associated with at least one swap chain and in windowed mode additional swap chains can be created to obtain multiple presentable surfaces.

In exclusive mode, a Direct3D application can control video scan out of the images in the frame buffer to the video monitor. The monitor can be driven in any of the display modes returned by `IDirect3D9` for the adapter.

The philosophy of the Direct3D object is to expose the capabilities of the hardware to the application programmer and let the application adapt to the hardware. The API does not, as a general rule, emulate features missing from the hardware. Differences in devices can be present based on device type and capability.

The COM objects returned by device properties have a reference added to them before they are returned. The application should call `Release` on all such objects when they are no longer needed, or a memory leak will result.

Interface 3.1: Summary of the `IDirect3DDevice9` interface.

IDirect3DDevice9

Read-Only Properties	
`GetAvailableTextureMem`	Available texture memory, in KB.
`GetBackBuffer`	Back buffer image surface interface
`GetCreationParameters`	Device creation parameters
`GetDeviceCaps`	Device capabilities structure
`GetDirect3D`	Creating `IDirect3D9` interface.
`GetDisplayMode`	Video display mode.
`GetFrontBufferData`	A copy of the front buffer.
`GetNumberOfSwapChains`	Number of swap chains.
`GetRasterStatus`	Raster display status.
`GetRenderTargetData`	Surface data in a render target.
`GetSwapChain`	Swap chain on the device.

Write-Only Properties	
`SetCursorPosition`	Cursor position.
`SetCursorProperties`	Cursor image and hot spot.
`SetDialogBoxMode`	GDI compatability flag.

Properties	
`GetClipPlane`	User-defined clipping planes.
`SetClipPlane`	
`GetClipStatus`	Clip status of rendered primitives.
`SetClipStatus`	
`GetCurrentTexturePalette`	Current texture palette index.
`SetCurrentTexturePalette`	

`GetDepthStencilSurface`	Depth surface interface
`SetDepthStencilSurface`	
`GetFVF`	Flexible vertex format of vertices.
`SetFVF`	
`GetGammaRamp`	Gamma correction lookup table.
`SetGammaRamp`	
`GetIndices`	Current index buffer.
`SetIndices`	
`GetLight`	Light definitions.
`SetLight`	
`GetLightEnable`	Light enabled flag.
`LightEnable`[1]	
`GetMaterial`	Primitive material properties.
`SetMaterial`	
`GetNPatchMode`	N-patch tessellation mode.
`SetNPatchMode`	
`GetPaletteEntries`	Texture palette entries.
`SetPaletteEntries`	
`GetPixelShader`	Pixel shader program handle.
`SetPixelShader`	
`GetPixelShaderConstantB`	Boolean constant registers.
`SetPixelShaderConstantB`	
`GetPixelShaderConstantF`	Floating-point constant registers.
`SetPixelShaderConstantF`	
`GetPixelShaderConstantI`	Integer constant registers.
`SetPixelShaderConstantI`	
`GetRenderState`	Pipeline control values.
`SetRenderState`	
`GetRenderTarget`	Target surface for rendered pixels.
`SetRenderTarget`	
`GetSamplerState`	Texture sampler control values.
`SetSamplerState`	
`GetScissorRect`	Scissor test rectangle.
`SetScissorRect`	
`GetSoftwareVertexProcessing`	Vertex processing control.
`SetSoftwareVertexProcessing`	
`GetStreamSource`	Source vertex data buffers.
`SetStreamSource`	
`GetStreamSourceFreq`	Stream source sampling frequency.
`SetStreamSourceFreq`	
`GetTexture`	Texture resources used by each stage.
`SetTexture`	
`GetTextureStageState`	Texture stage control values.
`SetTextureStageState`	

[1] This is a "set property" style method that isn't prefixed with `Set`.

GetTransform	Transformation matrices.
SetTransform	
GetVertexDeclaration	Vertex component declaration.
SetVertexDeclaration	
GetVertexShader	Vertex shader program.
SetVertexShader	
GetVertexShaderConstantB	Boolean constant registers.
SetVertexShaderConstantB	
GetVertexShaderConstantF	Floating-point constant registers.
SetVertexShaderConstantF	
GetVertexShaderConstantI	Integer constant registers.
SetVertexShaderConstantI	
GetViewport	Rendering viewport extent.
SetViewport	

Methods

BeginScene	Mark the start of a scene.
BeginStateBlock	Mark device state for capture.
Clear	Clears the current viewport on the device.
ColorFill	Fills a rectangular area with a color.
CreateAdditionalSwapChain	Create an additional swap chain on a windowed device.
CreateCubeTexture	Create a cube map texture sources.
CreateDepthStencilSurface	Create a depth/stencil surface resource.
CreateIndexBuffer	Create an index buffer resource.
CreateOffscreenPlainSurface	Create an off-screen surface resource.
CreatePixelShader	Create a pixel shader.
CreateQuery	Create a device query.
CreateRenderTarget	Create a render target resource.
CreateStateBlock	Create a state block.
CreateTexture	Create a texture resource.
CreateVertexBuffer	Create a vertex buffer resource.
CreateVertexDeclaration	Create a vertex declaration.
CreateVertexShader	Create a vertex shader.
CreateVolumeTexture	Create a volume texture resource.
DeletePatch	Destroy a cached patch tesselation.
DrawIndexedPrimitive	Draw indexed primitives from the current streams.
DrawIndexedPrimitiveUP	Draw indexed primitives from user data.
DrawPrimitive	Draw primitives from the current streams.
DrawPrimitiveUP	Draw primitives from user data.

`DrawRectPatch`	Rectangular higher order surface patch.
`DrawTriPatch`	Triangular higher order surface patch.
`EndScene`	Mark the end of a scene.
`EndStateBlock`	Capture device state changes.
`EvictManagedResources`	Flush managed resources from the device.
`MultiplyTransform`	Post-multiply onto a device transformation matrix.
`Present`	Presents a rendered image for display.
`ProcessVertices`	Process vertices in software.
`Reset`	Resets the device's display characteristics.
`ShowCursor`	Displays or hides the cursor.
`StretchRect`	Device memory blit with stretching.
`TestCooperativeLevel`	Checks exclusive ownership of the device.
`UpdateSurface`	System to device memory surface transfer.
`UpdateTexture`	System to device memory texture transfer.
`ValidateDevice`	Validate current device state.

Device Methods by Pipeline Section

Vertex Assembly	`BeginScene`
	`CreateIndexBuffer`
	`CreateVertexBuffer`
	`DeletePatch`
	`DrawIndexedPrimitive`
	`DrawIndexedPrimitiveUP`
	`DrawPrimitive`
	`DrawPrimitiveUP`
	`DrawRectPatch`
	`DrawTriPatch`
	`EndScene`
	`SetIndices`
	`SetNPatchMode`
	`SetRenderState`
	`SetStreamSource`
	`SetStreamSourceFreq`
	`SetVertexDeclaration`

. . . continued

Device Methods by Pipeline Section (continued)

Vertex Processing	`CreateVertexDeclaration`
	`CreateVertexShader`
	`LightEnable`
	`MultiplyTransform`
	`ProcessVertices`
	`SetClipPlane`
	`SetClipStatus`
	`SetFVF`
	`SetLight`
	`SetMaterial`
	`SetRenderState`
	`SetSoftwareVertexProcessing`
	`SetTransform`
	`SetVertexShader`
	`SetVertexShaderConstantB`
	`SetVertexShaderConstantF`
	`SetVertexShaderConstantI`
	`SetViewport`
Primitive Rasterization	`SetRenderState`
Pixel Processing	`CreatePixelShader`
	`MultiplyTransform`
	`SetCurrentTexturePalette`
	`SetPaletteEntries`
	`SetPixelShader`
	`SetPixelShaderConstantB`
	`SetPixelShaderConstantF`
	`SetPixelShaderConstantI`
	`SetRenderState`
	`SetSamplerState`
	`SetTexture`
	`SetTextureStageState`
	`SetTransform`

. . . continued

Device Methods by Pipeline Section (continued)	
Frame Buffer	`Clear`
	`ColorFill`
	`CreateAdditionalSwapChain`
	`CreateCubeTexture`
	`CreateDepthStencilSurface`
	`CreateOffscreenPlainSurface`
	`CreateRenderTarget`
	`CreateTexture`
	`CreateVolumeTexture`
	`EvictManagedResources`
	`GetAvailableTextureMem`
	`GetBackBuffer`
	`GetFrontBufferData`
	`GetNumberOfSwapChains`
	`GetRenderTargetData`
	`GetSwapChain`
	`Reset`
	`SetDepthStencilSurface`
	`SetDialogBoxMode`
	`SetRenderState`
	`SetRenderTarget`
	`SetScissorRect`
	`SetViewport`
	`StretchRect`
	`UpdateSurface`
	`UpdateTexture`
Video Scan Out	`GetDisplayMode`
	`GetRasterStatus`
	`Present`
	`SetCursorPosition`
	`SetCursorProperties`
	`SetGammaRamp`
	`ShowCursor`

Table 3.1: Device Methods by Pipeline Section

3.3 Capabilities

Advertising device capabilities is at the core of Direct3D's philosophy for exposing the device to an application. Direct3D doesn't provide software emulation for most of the features of the API. Rather than substitute a software implementation of a feature, Direct3D tells the application what features are supported by the device and lets the application adapt to the capacity of the hardware.

Table 3.2 shows device capabilities listed adjacent to their relevant pipeline stages. As a feature is discussed we will mention its relevant elements from the capabilities structure, but you may wish to familiarize yourself with the portions of the pipeline and their capabilities now.

Some capabilities are allowed to vary slightly once the device has been created. The `GetDeviceCaps` method on the device will return the capabilities of an actual device, while the `GetDeviceCaps` method on the Direct3D object returns the generic capabilities of a device.

```
HRESULT GetDeviceCaps(D3DCAPS9 *value);

typedef struct _D3DCAPS9
{
    D3DDEVTYPE DeviceType;
    UINT AdapterOrdinal;
    DWORD Caps;
    DWORD Caps2;
    DWORD Caps3;
    DWORD PresentationIntervals;
    DWORD CursorCaps;
    DWORD DevCaps;
    DWORD PrimitiveMiscCaps;
    DWORD RasterCaps;
    DWORD ZCmpCaps;
    DWORD SrcBlendCaps;
    DWORD DestBlendCaps;
    DWORD AlphaCmpCaps;
    DWORD ShadeCaps;
    DWORD TextureCaps;
    DWORD TextureFilterCaps;
    DWORD CubeTextureFilterCaps;
    DWORD VolumeTextureFilterCaps;
    DWORD TextureAddressCaps;
    DWORD VolumeTextureAddressCaps;
    DWORD LineCaps;
    DWORD MaxTextureWidth;
    DWORD MaxTextureHeight;
    DWORD MaxVolumeExtent;
    DWORD MaxTextureRepeat;
    DWORD MaxTextureAspectRatio;
    DWORD MaxAnisotropy;
    float MaxVertexW;
    float GuardBandLeft;
    float GuardBandTop;
    float GuardBandRight;
    float GuardBandBottom;
```

```
    float ExtentsAdjust;
    DWORD StencilCaps;
    DWORD FVFCaps;
    DWORD TextureOpCaps;
    DWORD MaxTextureBlendStages;
    DWORD MaxSimultaneousTextures;
    DWORD VertexProcessingCaps;
    DWORD MaxActiveLights;
    DWORD MaxUserClipPlanes;
    DWORD MaxVertexBlendMatrices;
    DWORD MaxVertexBlendMatrixIndex;
    float MaxPointSize;
    DWORD MaxPrimitiveCount;
    DWORD MaxVertexIndex;
    DWORD MaxStreams;
    DWORD MaxStreamStride;
    DWORD VertexShaderVersion;
    DWORD MaxVertexShaderConst;
    DWORD PixelShaderVersion;
    float PixelShader1xMaxValue;
    DWORD DevCaps2;
    float MaxNpatchTessellationLevel;
    DWORD Reserved5;
    UINT MasterAdapterOrdinal;
    UINT AdapterOrdinalInGroup;
    UINT NumberOfAdaptersInGroup;
    DWORD DeclTypes;
    DWORD NumSimultaneousRTs;
    DWORD StretchRectFilterCaps;
    D3DVSHADERCAPS2_0 VS20Caps;
    D3DPSHADERCAPS2_0 PS20Caps;
    DWORD VertexTextureFilterCaps;
    DWORD MaxVShaderInstructionsExecuted;
    DWORD MaxPShaderInstructionsExecuted;
    DWORD MaxVertexShader30InstructionSlots;
    DWORD MaxPixelShader30InstructionSlots;
} D3DCAPS9;
```

Device Capabilities by Pipeline Section

Vertex Assembly	**Dev Caps**:
	Draw Prim TL Vertex
	Quintic RT Patches
	RT Patches
	RT Patch Handle Zero
	TL Vertex System Memory

. . . continued

Device Capabilities by Pipeline Section (continued)	
	TL Vertex Video Memory
	Dev Caps 2:
	Adaptive Tess N Patch
	Adaptive Tess RT Patch
	D Map N Patch
	Stream Offset
	Vertex Elements Can Share Stream Offset
	Decl Types
	FVF Caps
	Max Primitive Count
	Max Streams
	Max Stream Stride
	Max Vertex Index
	Max N Patch Tessellation Level
Vertex Processing	**Dev Caps**:
	HW Transform And Light
	Pure Device
	Max Active Lights
	Max User Clip Planes
	Max Vertex Blend Matrices
	Max Vertex Blend Matrix Index
	Max Vertex Shader 30 Instruction Slots
	Max Vertex Shader Const
	Max V Shader Instructions Executed
	Primitive Misc Caps:
	Clip Plane Scaled Points
	Clip TL Verts
	Fog Vertex Clamped
	Raster Caps:
	Fog Range
	Fog Vertex
	Vertex Processing Caps
	Vertex Shader Version
	Vertex Texture Filter Caps
	VS 20 Caps
Primitive Processing	**Max User Clip Planes**
	Primitive Misc Caps:
	Clip Plane Scaled Points
	Clip TL Verts
	Cull CW
	Cull CCW
	Cull None
Primitive Rasterization	**Dev Caps**:
	HW Rasterization
	Pure Hardware
	Extents Adjust
	Guard Band Left
	Guard Band Top

. . . continued

Device Capabilities by Pipeline Section (continued)

	Guard Band Right
	Guard Band Bottom
	Line Caps:
	Antialias
	Texture
	Max Point Size
	Max Vertex W
	Raster Caps:
	Color Perspective
	Shade Caps
	Texture Caps:
	Projected
Pixel Processing	**Caps 2**:
	Can Auto Gen Mipmap
	Cube Texture Filter Caps
	Dev Caps:
	Separate Texture Memories
	Texture Non Local Vid Mem
	Texture System Memory
	Texture Video Memory
	Max Anisotropy
	Max Pixel Shader 30 Instruction Slots
	Max P Shader Instructions Executed
	Max Simultaneous Textures
	Max Texture Aspect Ratio
	Max Texture Blend Stages
	Max Texture Height
	Max Texture Repeat
	Max Texture Width
	Max Volume Extent
	Pixel Shader Version
	Pixel Shader 1x Max Value
	Primitive Misc Caps:
	Per Stage Constant
	TSS Arg Temp
	PS 20 Caps
	Raster Caps:
	Anisotropy
	Mipmap LOD Bias
	Texture Address Caps
	Texture Caps
	Texture Filter Caps
	Texture Op Caps
	Volume Texture Filter Caps
Frame Buffer	**Alpha Cmp Caps**
	Caps 3:
	Alpha Fullscreen Flip Or Discard
	Copy To System Mem

. . . continued

Device Capabilities by Pipeline Section (continued)	
	Copy To Vid Mem
	Dest Blend Caps
	Line Caps:
	Alpha Cmp
	Blend
	Fog
	Z Test
	Num Simultaneous RTs
	Primitive Misc Caps:
	Blend Op
	Color Write Enable
	Independent Write Masks
	Mask Z
	MRT Independent Bit Depths
	MRT Post Pixel Shader Blending
	Separate Alpha Blend
	Raster Caps:
	Dither
	Fog Table
	Multisample Toggle
	Scissor Test
	W Buffer
	W Fog
	Z Bufferless HSR
	Z Fog
	Z Test
	Src Blend Caps
	Stencil Caps
	Stretch Rect Filter Caps
	Z Cmp Caps
Video Scan Out	**Adapter Ordinal In Group**
	Caps:
	Read Scan Line
	Caps 2:
	Can Calibrate Gamma
	Full Screen Gamma
	Caps 3:
	Linear To sRGB Presentation
	Cursor Caps
	Dev Caps:
	Can Render After Flip
	Master Adapter Ordinal
	Number Of Adapters In Group
	Presentation Intervals

Table 3.2: Device capabilities organized by pipeline section. `D3DCAPS9` member names are written as word phrases and bit flag names are written without their prefix in mixed case, indented beneath the containing structure member.

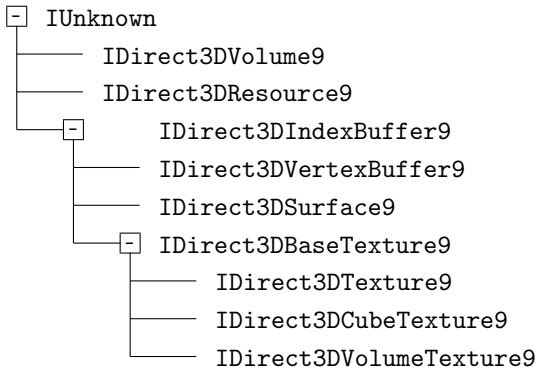

Figure 3.1: Resource interface hierarchy as a tree view; subtrees denote derivation. All COM interfaces derive from `IUnknown`.

3.4 Resources

The device works together with resource objects to perform scene rendering. Resource objects are used as containers for the scene data rendered by the device such as primitive vertices, indices into vertex arrays, surface textures and volumes. The resource interface hierarchy is shown in figure 3.1

Two and three dimensional textures expose their contents as collections of surfaces and volumes, respectively. The back buffer, depth/stencil buffer and render target properties of the device are also exposed as surfaces. Volume objects do not inherit from `IDirect3DResource9` and therefore do not participate in the resource management exposed by this interface.

The remaining resource objects all inherit from `IDirect3DResource9`. Additionally, all the texture resources inherit from `IDirect3DBaseTexture9`, covered in chapter 11.

3.5 `IDirect3DResource9`

`IDirect3DResource9` is the base interface for all resource objects and exposes resource memory management features for managed resources. The interface is summarized in interface 3.2. Managing a resource governs when the resource is promoted from system memory to memory accessible by the device and when the resource is discarded from device memory. Only resources created in `D3DPOOL_MANAGED` are managed.

Managed resources are assigned an unsigned integer priority, with higher priority taking precedence so that resources with a lower priority are discarded from device memory first. Non-managed resources always return a priority of zero. Within the same priority, Direct3D uses a least-recently used strategy to discard "old" resources in preference to newly created resources. When the application attempts to use more resources than the card can hold while rendering

a scene, Direct3D switches to a most-recently used first strategy for discarding resources. This helps prevent thrashing of resources in and out of device memory.

An application can explicitly cause the resource manager to discard memory consumed by managed resources with `EvictManagedResources`. The released device memory will not necessarily be contiguous. Allocating an object may still fail due insufficient memory even though the resource manager freed enough total space.

```
HRESULT EvictManagedResources();
```

Resources in `D3DPOOL_DEFAULT` are not managed and are never discarded from device memory by the resource manager. An application can perform its own resource management by creating resources in the default pool only and manually allocating, copying and releasing resources from the device memory. It is recommended that you use Direct3D's resource management until you have measured a bottleneck to be located within the resource management.

Because resources in the default pool are not discarded, the best results from Direct3D's resource management algorithms will be obtained if the application creates all resources in `D3DPOOL_DEFAULT` before resources in `D3DPOOL_MANAGED`. If you need to allocate new resources in the default pool after managed resources have been loaded, you should evict all managed resources before allocating new resources in the default pool.

Resources in pool `D3DPOOL_SYSTEMMEM` are never located in device memory, so they do not participate in resource management. However, resources in `D3DPOOL_SYSTEMMEM` and `D3DPOOL_MANAGED` contribute to the paging load of the application since they reside in the memory space of the application's process.

When loading textures into a device, `GetAvailableTextureMem` can be called to obtain an estimate of the available texture memory. This estimate can be used to adjust the detail of texture resources used by the application. However, its best to use this value only as a rough guideline and not as an exact measurement; the runtime can manage device memory and it is generally best to leave it to the runtime. See chapter 11 for more on texture resources.

```
UINT GetAvailableTextureMem();
```

Interface 3.2: Summary of the `IDirect3DResource9` interface.

IDirect3DResource9

Read-Only Properties	
`GetDevice`	The owning device.
`GetType`	The resource type.

Properties

`GetPriority`	Management priority of the resource.
`SetPriority`	
`GetPrivateData`	Private data associated with the resource.
`SetPrivateData`	

Methods

`FreePrivateData`	Frees associated private data memory.
`PreLoad`	Forces a managed resource into device memory.

```
interface IDirect3DResource9 : IUnknown
{
 //---------------------------------------------------------------
 // read-only properties
         HRESULT GetDevice(IDirect3DDevice9 **value);
 D3DRESOURCETYPE GetType();

 //---------------------------------------------------------------
 // read/write properties
           DWORD GetPriority();
           DWORD SetPriority(DWORD value);
         HRESULT GetPrivateData(REFGUID data_guid,
                     void *value,
                     DWORD *size);
         HRESULT SetPrivateData(REFGUID data_guid,
                     const void *value,
                     DWORD size,
                     DWORD flags);

 //---------------------------------------------------------------
 // methods
         HRESULT FreePrivateData(REFGUID data_guid);
            void PreLoad();
};
```

The `GetDevice` method returns the device with which this resource is associated. Resources cannot be shared across devices. The type of the resource is returned by `GetType` and can be used to call `QueryInterface` with the appropriate IID to obtain a more specialized interface for the resource.

For instance, the following function returns the `IDirect3DTexture9` interface associated with a texture resource or NULL otherwise. It is the caller's responsibility to call `Release` on the returned interface when finished. This is one of the few times you will call `QueryInterface` on a Direct3D object. Here we are using `QueryInterface` to perform a type-safe "downcast" from the base class `IDirect3DResource9` to the derived class `IDirect3DTexture9`.

```
IDirect3DTexture9 *
resource_texture(IDirect3DResource9 *resource)
{
    if (D3DRTYPE_TEXTURE != resource->GetType())
    {
        return NULL;
    }

    IDirect3DTexture9 *texture = NULL;
    THR(resource->
        QueryInterface(IID_IDirect3DTexture9, &texture));
    return texture;
}
```

`GetPrivateData` and `SetPrivateData` allow an application to associate its own arbitrary chunks of data with any Direct3D resource.[2] Each distinct item of private data is identified by a GUID. You can generate GUIDs for data items used by your application by a tool such as `GUIDGEN.EXE` in the Platform SDK.

Private data is passed by value to the device. When private data is set on a device, the data is copied from the supplied pointer into a block of memory allocated by the device. When private data is gotten from a device, the data is copied from the device's block of memory into the block supplied by the caller.

All private data associated with a resource is freed when the associated resource itself is freed. If private data is already set for a particular GUID and data is set on that GUID again, the existing private data is freed before the new private data is copied into a newly allocated block. You can explicitly free private data with the `FreePrivateData` method.

When the private data is an interface pointer, `IDirect3DResource9` can be instructed to manage the interface pointer appropriately by passing `D3DSPD_IUNKNOWN` for the `flags` argument of `SetPrivateData`. After the interface pointer is copied into the device, the device calls `AddRef` on it. Before the interface pointer is freed, `Release` is called on it. Adding a reference to the supplied interface ensures that the interface pointer remains valid for the lifetime of the private data.

```
#define D3DSPD_IUNKNOWN 0x00000001L
```

Like many Win32 functions, `GetPrivateData` returns the data as a sized block of memory. You first call the function with a `NULL` data pointer to obtain the size of the block, allocate a block big enough, and then call the function again with the pointer to the block of memory. In this example, a `std::vector` is used to obtain the storage. You can avoid the first step if you previously set the private data on the resource and you already know its size.

[2] `IDirect3DSurface9`, although it does not derive from `IDirect3DResource9` has identical methods for private data.

```
// get buffer size
DWORD size = 0;
THR(resource->GetPrivateData(data_id, NULL, &size));
// allocate buffer
std::vector<BYTE> data(size);
// fill buffer contents
THR(resource->GetPrivateData(data_id, &data[0], &size));
```

3.6 Destroying a Device

When an application is finished with a device, the device can be released by calling the `Release` from `IUnknown`. This decrements the reference count on the device object. When no outstanding references exist to a COM object, it can be safely destroyed and its memory released.

Methods and functions in Direct3D that create COM objects, such as `CreateDevice`, add a reference to the object for the caller before they return the interface pointer. The application must release these objects when they are no longer needed to avoid a memory leak.

3.7 Miscellaneous Properties

A few device properties are not directly associated with any particular portion of the pipeline and describe the device in general. The `GetCreationParameters` method returns the parameters used to create the device in a `D3DDEVICE_CREATION_PARAMETERS` structure.

```
HRESULT GetCreationParameters(
            D3DDEVICE_CREATION_PARAMETERS *value);
HRESULT GetDirect3D(IDirect3D9 **value);

typedef struct _D3DDEVICE_CREATION_PARAMETERS
{
    UINT       AdapterOrdinal;
    D3DDEVTYPE DeviceType;
    HWND       hFocusWindow;
    DWORD      BehaviorFlags;
} D3DDEVICE_CREATION_PARAMETERS;
```

The instance of `IDirect3D9` that created the device can be obtained with the `GetDirect3D` method. You may need this to re-enumerate display modes on the adapter if you didn't cache this information.

3.8 Device Queries

Device queries allow you to obtain information from the driver layer of the device. The two main uses for driver queries are for obtaining rendering statistics and event notifications from the device. A query is represented as a COM object. As long as the query object and its associated device exist you can issue the query to the device to obtain fresh information. To create a query call the `CreateQuery` method with an enumeration identifying the type of information you wish to query.

```
typedef enum _D3DQUERYTYPE {
    D3DQUERYTYPE_BANDWIDTHTIMINGS  = 17,
    D3DQUERYTYPE_CACHEUTILIZATION  = 18,
    D3DQUERYTYPE_EVENT             = 8,
    D3DQUERYTYPE_INTERFACETIMINGS  = 14,
    D3DQUERYTYPE_OCCLUSION         = 9,
    D3DQUERYTYPE_PIPELINETIMINGS   = 13,
    D3DQUERYTYPE_PIXELTIMINGS      = 16,
    D3DQUERYTYPE_RESOURCEMANAGER   = 5,
    D3DQUERYTYPE_TIMESTAMP         = 10,
    D3DQUERYTYPE_TIMESTAMPDISJOINT = 11,
    D3DQUERYTYPE_TIMESTAMPFREQ     = 12,
    D3DQUERYTYPE_VCACHE            = 4,
    D3DQUERYTYPE_VERTEXSTATS       = 6,
    D3DQUERYTYPE_VERTEXTIMINGS     = 15
} D3DQUERYTYPE;

HRESULT CreateQuery(D3DQUERYTYPE kind, IDirect3DQuery9 **result);
```

The `D3DQUERYTYPE` enumeration gives the possible kinds of queries: vertex cache description queries, resource manager statistics queries, vertex statistics queries, event queries, occlusion queries, timestamp queries, timing queries and cache utilization queries. The methods of the query interface are summarized in interface 3.3.

Interface 3.3: Summary of the `IDirect3DQuery9` interface.

IDirect3DQuery9

Read-Only Properties	
`GetData`	The query result data.
`GetDataSize`	The size of the query result data.
`GetDevice`	The associated device.
`GetType`	The query type.

Methods	
`Issue`	Issues a query to the device.

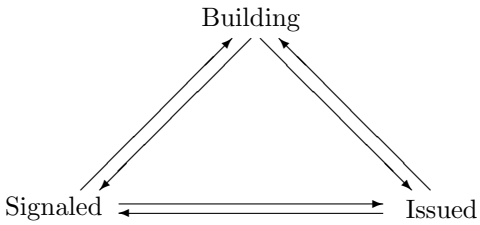

Figure 3.2: A query object transits between Building, Issued and Signaled states through calls to the `Issue` method by the application or by the device driver.

```
interface IDirect3DQuery9 : IUnknown
{
 //------------------------------------------------------------
 // read-only properties
      HRESULT GetDevice(IDirect3DDevice9 **value);
 D3DQUERYTYPE GetType();
      HRESULT GetData(void *data, DWORD size, DWORD flags);
        DWORD GetDataSize();

 //------------------------------------------------------------
 // methods
      HRESULT Issue(DWORD flags);
};
```

A query exists in one of three states: signaled, building, or issued. The transitions between these states are shown in figure 3.2. The `Issue` method is used by the application to signal a state transition on the query. The device driver can also change the state of a query when it has returned the data requested by the query. Queries are sent to the device in the rendering command stream and the results are available after all the geometry before the query has been processed. The `flags` argument to `Issue` is used to signal the beginning or end of a query.

```
#define D3DISSUE_END   (1 << 0)
#define D3DISSUE_BEGIN (1 << 1)
```

The query will report the results for primitives issued between the beginning of the query and the end of the query. Vertex cache, resource manager statistics and event queries have an implicit beginning and the application issues the end of the query. The implicit beginnings of queries are listed in table 3.3. The statistics queries return data collected since the last call to `Present`, while the event and vertex cache queries return data since the creation of the device.

Query Type	Beginning
Vertex Cache	`CreateDevice`
Resource Manager Statistics	`Present`
Vertex Statistics	`Present`
Event	`CreateDevice`

Table 3.3: Implicit Query Beginnings

Query Type	Data Type
Bandwidth Timings	`D3DDEVINFO_D3D9BANDWIDTHTIMINGS`
Cache Utilization	`D3DDEVINFO_D3D9CACHEUTILIZATION`
Event	`BOOL`
Interface Timings	`D3DDEVINFO_D3D9INTERFACETIMINGS`
Occlusion	`DWORD`
Pipeline Timings	`D3DDEVINFO_D3D9PIPELINETIMINGS`
Pixel Shader Timings	`D3DDEVINFO_D3D9STAGETIMINGS`
Resource Manager Statistics	`D3DDEVINFO_RESOURCEMANAGER`
Timestamp	`UINT64`
Timestamp Disjoint	`BOOL`
Timestamp Frequency	`UINT64`
Vertex Cache	`D3DDEVINFO_VCACHE`
Vertex Shader Timings	`D3DDEVINFO_D3D9STAGETIMINGS`
Vertex Statistics	`D3DDEVINFO_D3DVERTEXSTATS`

Table 3.4: Query Data Types

For occlusion, disjoint timestamp and timing queries, the application explicitly specifies the beginning and end of the query.

To get the results of a query, call its `GetDataSize` and `GetData` methods to obtain a copy of the query data. All queries return data and expect a suitably sized data structure to be passed to `GetData`. The data types of the query data are given in table 3.4. The `size` argument should match the size of the data structure, in bytes.

`GetData` returns `S_OK` when data is available and `S_FALSE` when the query data is not yet available. Both of these values are `HRESULT` values indicating success. A lost device cannot return query data and `GetData` returns `D3DERR_DEVICELOST`.

The `flags` argument to `GetData` lets you synchronously flush the command queue to the device.

```
#define D3DGETDATA_FLUSH (1 << 0)
```

The event query can be used to inform the application when a particular point in the command stream has been processed by the driver. The query data is a boolean indicating if the location of the event query in the command stream has been processed.

Occlusion queries return the number of pixels that passed the depth test for primitives rendered between the begin and end of the query. A value of zero indicates that the group of primitives is completely obscured from the camera by foreground primitives. An application can use this information to cull occluded objects from the scene.

You can probe a device for query support by attempting to create a query object with a `NULL` pointer for the result. `CreateQuery` will fail if the query is not supported and will succeed if the query is supported. In addition, the vertex statistics and resource manager queries are only supported with the debug developer runtime and are not supported with the retail runtime.

3.8.1 Resource Manager Statistics Queries

The resource manager statistics query returns a `D3DDEVINFO_RESOURCEMANAGER` structure, which contains an array of `D3DRESOURCESTATS` structures, one for each resource type. There are `D3DRTYPECOUNT` resource types.

```
#define D3DRTYPECOUNT (D3DRTYPE_INDEXBUFFER+1)

typedef struct _D3DDEVINFO_RESOURCEMANAGER
{
    D3DRESOURCESTATS stats[D3DRTYPECOUNT];
} D3DDEVINFO_RESOURCEMANAGER;
```

The `D3DRESOURCESTATS` structure has statistics since the last call to `Present` on the device, as well as statistics kept since the device was last `Reset`. The statistics for each resource indicate the performance of the resource in `D3DPOOL_MANAGED`. Resources in other memory pools are not reported in these statistics.

```
typedef struct _D3DRESOURCESTATS
{
    // Data collected since last Present()
    BOOL    bThrashing;
    DWORD   ApproxBytesDownloaded;
    DWORD   NumEvicts;
    DWORD   NumVidCreates;
    DWORD   LastPri;
    DWORD   NumUsed;
    DWORD   NumUsedInVidMem;

    // Persistent data
    DWORD   WorkingSet;
    DWORD   WorkingSetBytes;
    DWORD   TotalManaged;
    DWORD   TotalBytes;
} D3DRESOURCESTATS;
```

The persistent data describes the device memory working set and some total statistics. The `WorkingSet` member is the number of resource objects in video memory and `WorkingSetBytes` is the size of the working set. The `TotalManaged` member gives the number of managed resource objects and `TotalBytes` are their total size.

The frame related data describes the behavior of the resource manager for the last frame. The `bThrashing` member is set if the resource manager is "thrashing". Thrashing occurs when the resource manager is constantly streaming resources into video memory. For instance, suppose you have a large number of objects in the scene, each using one of a set of textures. If the objects are drawn sorted by texture, then it is more likely that the necessary textures will be already loaded in device memory. If the objects are drawn without regard to texture, then the resource manager may spend lots of time thrashing back and forth between different textures in the working set.

The `ApproxBytesDownloaded`, `NumEvicts` and `NumVidCreates` members describe the resource consumption behavior of the resource manager since the last call to `Present`. The `LastPri` member gives the priority of the last object that was evicted from device memory. The `NumUsed` and `NumUsedInVidMem` describe how many resource objects were set on the device and how many were already present in device memory, respectively.

3.8.2 Vertex Statistics Queries

Statistics on vertex processing since the last call to `Present` are also reported by the device. The total number of triangles drawn since the last `Present` and the number of additional triangles introduced through clipping are reported in the `D3DDEVINFO_D3DVERTEXSTATS` structure.

```
typedef struct _D3DDEVINFO_D3DVERTEXSTATS
{
    DWORD NumRenderedTriangles;
    DWORD NumExtraClippingTriangles;
} D3DDEVINFO_D3DVERTEXSTATS;
```

3.8.3 Vertex Cache Queries

The vertex cache query returns information about the size of the hardware vertex cache on the device. The vertex cache is a memory cache close to the GPU that avoids accesses to vertex memory for a small number of recently used vertices. The query returns a `D3DDEVINFO_VCACHE` structure.

```
typedef struct _D3DDEVINFO_VCACHE {
    DWORD   Pattern;
    DWORD   OptMethod;
    DWORD   CacheSize;
    DWORD   MagicNumber;
} D3DDEVINFO_VCACHE, *LPD3DDEVINFO_VCACHE;
```

The `Pattern` member must be the four-character code `CACH`. When the `OptMethod` member is zero, applications should use the longest possible triangle strips to maximize vertex cache coherence. Otherwise, the value is one to indicate that applications should optimize for a vertex cache with `CacheSize` entries. The `MagicNumber` member determines when to restart triangle strips when the `OptMethod` member is one.

3.8.4 PIX Related Queries

PIX is a performance measurement tool for DirectX applications. The remaining query types return data for performance measurements with tools like PIX. A driver may not support all of these queries, or all the members within the structure returned by a particular query. If the values returned are zero, you should ignore the measurement.

The cache utilization query returns the cache hit rate for the texel cache and the vertex cache.

```
typedef struct _D3DDEVINFO_D3D9CACHEUTILIZATION
{
    float TextureCacheHitRate;
    float PostTransformVertexCacheHitRate;
} D3DDEVINFO_D3D9CACHEUTILIZATION;
```

All of the timing queries return information indicating the portion of the query time spent in various portions of the pipeline. The measurements reflect the percentages between the time of the beginning and end of the query.

```
typedef struct _D3DDEVINFO_D3D9BANDWIDTHTIMINGS
{
    float MaxBandwidthUtilized;
    float FrontEndUploadMemoryUtilizedPercent;
    float VertexRateUtilizedPercent;
    float TriangleSetupRateUtilizedPercent;
    float FillRateUtilizedPercent;
} D3DDEVINFO_D3D9BANDWIDTHTIMINGS;


typedef struct _D3DDEVINFO_D3D9INTERFACETIMINGS
{
    float WaitingForGPUToUseApplicationResourceTimePercent;
    float WaitingForGPUToAcceptMoreCommandsTimePercent;
    float WaitingForGPUToStayWithinLatencyTimePercent;
    float WaitingForGPUExclusiveResourceTimePercent;
    float WaitingForGPUOtherTimePercent;
} D3DDEVINFO_D3D9INTERFACETIMINGS;
```

```
typedef struct _D3DDEVINFO_D3D9PIPELINETIMINGS
{
    float VertexProcessingTimePercent;
    float PixelProcessingTimePercent;
    float OtherGPUProcessingTimePercent;
    float GPUIdleTimePercent;
} D3DDEVINFO_D3D9PIPELINETIMINGS;
```

The vertex shader timing and pixel shader timing queries both return the information in the D3DDEVINFO_D3D9STAGETIMINGS structure.

```
typedef struct _D3DDEVINFO_D3D9STAGETIMINGS
{
    float MemoryProcessingPercent;
    float ComputationProcessingPercent;
} D3DDEVINFO_D3D9STAGETIMINGS;
```

3.9 Device States

The device object's properties control the behavior of the rendering pipeline while its methods supply data for the pipeline to render. The render, sampler and texture stage state properties of the device control many aspects of the pipeline's behavior and are summarized by pipeline section in table 3.5.

Device States by Pipeline Section

Vertex Assembly	RS Patch Edge Style	RS Adaptive Tess X, Y, Z, W
	RS Position Degree	RS Enable Adaptive Tessellation
	RS Normal Degree	RS Max Tessellation Level
	RS Color Vertex	RS Min Tessellation LEvel
	RS Point Size	SS D Map Offset
Vertex Processing	RS Tween Factor	RS Clipping
	RS Vertex Blend	RS Indexed Vertex Blend Enable
	TSS Tex Coord Index	RS Ambient Material Source
	RS Fog Vertex Mode	RS Diffuse Material Source
	RS Range Fog Enable	RS Specular Material Source
	RS Ambient	RS Emissive Material Source
	RS Local Viewer	RS Lighting
	RS Specular Enable	RS Normalize Normals
	RS Shade Mode	TSS Texture Transform Flags
	RS Cull Mode	RS Clip Plane Enable
Pixel Processing		SS Border Color
	RS Fill Mode	SS Mag Filter
	RS Last Pixel	SS Min Filter
		SS Mip Filter
	RS Specular Enable	SS Mip Map LOD Bias

. . . continued

Device States by Pipeline Section (continued)		
	RS Texture Factor	SS Max Mip Level
	RS Wrap $0-15$	SS Max Anisotropy
	TSS Tex Coord Index	TSS Texture Transform Flags
	SS Address U, V, W	TSS Color Arg $0-2$
	TSS Color Op	TSS Constant
	TSS Alpha Arg $0-2$	RS Fog Table Mode
	TSS Alpha Op	RS Fog Density
	TSS Result Arg	RS Fog End
	TSS Bump Env Mat $00-11$	RS Fog Start
	TSS Bump Env L Scale	RS Fog Color
	TSS Bump Env L Offset	RS Fog Enable
	RS Depth Bias	RS Slope Scale Depth Bias
	SS sRGB Texture	RS Antialiased Line Enable
Frame	RS Alpha Ref	RS Alpha Test Enable
Buffer	RS Alpha Func	RS Z Enable
	RS Z Func	
	RS Stencil Enable	RS Stencil Func
	RS Stencil Ref	RS Stencil Mask
	RS Stencil Fail	RS Stencil Z Fail
	RS Stencil Pass	RS Alpha Blend Enable
	RS Src Blend	RS Dest Blend
	RS Blend Op	RS Dither Enable
	RS Color Write Enable	RS Stencil Write Mask
	RS Z Write Enable	RS Multi Sample Antialias
	RS Multi Sample Mask	RS Scissor Test Enable
	RS Src Blend Alpha	RS Separate Alpha Blend Enable
	RS Dest Blend Alpha	RS Blend Op Alpha
	RS Blend Factor	RS CCW Stencil Fail
	RS CCW Stencil Z Fail	RS CCW Stencil Pass
	RS sRGB Write Enable	RS CCW Stencil Func
	SS Element Index	RS Color Write Enable $1-3$
		RS Two Sided Stencil Mode

Table 3.5: Render and texture stage states organized by pipeline section. "RS", "SS" and "TSS" denote render, sampler and texture stage states. There are no states that affect video scan out.

Render states, sampler states and texture stage states are 32-bit quantities having a name and a value. The names are given by the `D3DRENDERSTATETYPE`, `D3DSAMPLERSTATETYPE` and `D3DTEXTURESTAGESTATETYPE` enumerations. These properties are manipulated through the `GetRenderState`, `SetRenderState`, `GetSamplerState`, `SetSamplerState`, `GetTextureStageState`,

and `SetTextureStageState` methods. The details of sampler and texture stage states are discussed in chapter 11. The details of individual render states are discussed throughout the book for each section of the pipeline.

```
HRESULT GetRenderState(D3DRENDERSTATETYPE kind,
            DWORD *value);
HRESULT SetRenderState(D3DRENDERSTATETYPE kind,
            DWORD value);
HRESULT GetSamplerState(DWORD stage,
            D3DSAMPLERSTATETYPE kind,
            DWORD *value);
HRESULT SetSamplerState(DWORD stage,
            D3DSAMPLERSTATETYPE kind,
            DWORD value);
HRESULT GetTextureStageState(DWORD stage,
            D3DTEXTURESTAGESTATETYPE kind,
            DWORD *value);
HRESULT SetTextureStageState(DWORD stage,
            D3DTEXTURESTAGESTATETYPE kind,
            DWORD value);


typedef enum _D3DRENDERSTATETYPE {
    D3DRS_ALPHABLENDENABLE          = 27,
    D3DRS_ALPHAFUNC                 = 25,
    D3DRS_ALPHAREF                  = 24,
    D3DRS_ALPHATESTENABLE           = 15,
    D3DRS_AMBIENT                   = 139,
    D3DRS_AMBIENTMATERIALSOURCE     = 147,
    D3DRS_BLENDOP                   = 171,
    D3DRS_CLIPPING                  = 136,
    D3DRS_CLIPPLANEENABLE           = 152,
    D3DRS_COLORWRITEENABLE          = 168,
    D3DRS_COLORVERTEX               = 141,
    D3DRS_CULLMODE                  = 22,
    D3DRS_DEBUGMONITORTOKEN         = 165,
    D3DRS_DESTBLEND                 = 20,
    D3DRS_DIFFUSEMATERIALSOURCE     = 145,
    D3DRS_DITHERENABLE              = 26,
    D3DRS_EDGEANTIALIAS             = 40,
    D3DRS_EMISSIVEMATERIALSOURCE    = 148,
    D3DRS_FILLMODE                  = 8,
    D3DRS_FOGCOLOR                  = 34,
    D3DRS_FOGDENSITY                = 38,
    D3DRS_FOGENABLE                 = 28,
    D3DRS_FOGEND                    = 37,
    D3DRS_FOGSTART                  = 36,
```

```
D3DRS_FOGTABLEMODE              = 35,
D3DRS_FOGVERTEXMODE             = 140,
D3DRS_INDEXEDVERTEXBLENDENABLE  = 167,
D3DRS_LASTPIXEL                 = 16,
D3DRS_LIGHTING                  = 137,
D3DRS_LINEPATTERN               = 10,
D3DRS_LOCALVIEWER               = 142,
D3DRS_NORMALIZENORMALS          = 143,
D3DRS_MULTISAMPLEANTIALIAS      = 161,
D3DRS_MULTISAMPLEMASK           = 162,
D3DRS_NORMALORDER               = 173,
D3DRS_PATCHEDGESTYLE            = 163,
D3DRS_PATCHSEGMENTS             = 164,
D3DRS_POINTSCALE_A              = 158,
D3DRS_POINTSCALE_B              = 159,
D3DRS_POINTSCALE_C              = 160,
D3DRS_POINTSCALEENABLE          = 157,
D3DRS_POINTSIZE                 = 154,
D3DRS_POINTSIZE_MAX             = 166,
D3DRS_POINTSIZE_MIN             = 155,
D3DRS_POINTSPRITEENABLE         = 156,
D3DRS_POSITIONORDER             = 172,
D3DRS_RANGEFOGENABLE            = 48,
D3DRS_SHADEMODE                 = 9,
D3DRS_SOFTWAREVERTEXPROCESSING  = 153,
D3DRS_SPECULARENABLE            = 29,
D3DRS_SPECULARMATERIALSOURCE    = 146,
D3DRS_SRCBLEND                  = 19,
D3DRS_STENCILENABLE             = 52,
D3DRS_STENCILFAIL               = 53,
D3DRS_STENCILFUNC               = 56,
D3DRS_STENCILMASK               = 58,
D3DRS_STENCILPASS               = 55,
D3DRS_STENCILREF                = 57,
D3DRS_STENCILWRITEMASK          = 59,
D3DRS_STENCILZFAIL              = 54,
D3DRS_TEXTUREFACTOR             = 60,
D3DRS_TWEENFACTOR               = 170,
D3DRS_VERTEXBLEND               = 151,
D3DRS_WRAP0                     = 128,
D3DRS_WRAP1                     = 129,
D3DRS_WRAP2                     = 130,
D3DRS_WRAP3                     = 131,
D3DRS_WRAP4                     = 132,
D3DRS_WRAP5                     = 133,
D3DRS_WRAP6                     = 134,
```

```
    D3DRS_WRAP7                     = 135,
    D3DRS_ZBIAS                     = 47,
    D3DRS_ZENABLE                   = 7,
    D3DRS_ZFUNC                     = 23,
    D3DRS_ZVISIBLE                  = 30,
    D3DRS_ZWRITEENABLE              = 14
} D3DRENDERSTATETYPE;


%
typedef enum _D3DSAMPLERSTATETYPE
{
    D3DSAMP_ADDRESSU       = 1,
    D3DSAMP_ADDRESSV       = 2,
    D3DSAMP_ADDRESSW       = 3,
    D3DSAMP_BORDERCOLOR    = 4,
    D3DSAMP_DMAPOFFSET     = 13,
    D3DSAMP_ELEMENTINDEX   = 12,
    D3DSAMP_MAGFILTER      = 5,
    D3DSAMP_MAXANISOTROPY  = 10,
    D3DSAMP_MAXMIPLEVEL    = 9,
    D3DSAMP_MINFILTER      = 6,
    D3DSAMP_MIPFILTER      = 7,
    D3DSAMP_MIPMAPLODBIAS  = 8,
    D3DSAMP_SRGBTEXTURE    = 11
} D3DSAMPLERSTATETYPE;


typedef enum _D3DTEXTURESTAGESTATETYPE
{
    D3DTSS_ADDRESSU              = 13,
    D3DTSS_ADDRESSV              = 14,
    D3DTSS_ADDRESSW              = 25,
    D3DTSS_ALPHAARG0             = 27,
    D3DTSS_ALPHAARG1             = 5,
    D3DTSS_ALPHAARG2             = 6,
    D3DTSS_ALPHAOP               = 4,
    D3DTSS_BORDERCOLOR           = 15,
    D3DTSS_BUMPENVLOFFSET        = 23,
    D3DTSS_BUMPENVLSCALE         = 22,
    D3DTSS_BUMPENVMAT00          = 7,
    D3DTSS_BUMPENVMAT01          = 8,
    D3DTSS_BUMPENVMAT10          = 9,
    D3DTSS_BUMPENVMAT11          = 10,
    D3DTSS_COLORARG0             = 26,
    D3DTSS_COLORARG1             = 2,
    D3DTSS_COLORARG2             = 3,
```

```
    D3DTSS_COLOROP               = 1,
    D3DTSS_MAGFILTER             = 16,
    D3DTSS_MAXANISOTROPY         = 21,
    D3DTSS_MAXMIPLEVEL           = 20,
    D3DTSS_MINFILTER             = 17,
    D3DTSS_MIPFILTER             = 18,
    D3DTSS_MIPMAPLODBIAS         = 19,
    D3DTSS_RESULTARG             = 28,
    D3DTSS_TEXCOORDINDEX         = 11,
    D3DTSS_TEXTURETRANSFORMFLAGS = 24
} D3DTEXTURESTAGESTATETYPE;
```

All state values are 32-bit sized quantities and the API treats these quantities as `DWORD`s. However, each render, sampler or texture stage state is associated with a 32-bit datatype that is not necessarily a `DWORD`. The data type associated with the device render and texture stage states are summarized in table 3.6. Most of these types are enumeration values, but some of the state values are floating-point parameters. Since Direct3D uses IEEE single-precision floating-point values, they are also 32-bit quantities, but the property methods take `DWORD` parameters and the floating-point values must be converted. The technique used is to reinterpret the 32-bit memory pattern as needed with `reinterpret_cast`. The functions `float_dword` and `dword_float` in listing 3.1 show how to reinterpret the bit patterns.

Device State Data Types

Device State	Data Type
RS Adaptive Tess X, Y, Z, W	`float`
RS Alpha Blend Enable	`BOOL`[1]
RS Alpha Func	`D3DCMPFUNC`
RS Alpha Ref	$[0, 255]$
RS Alpha Test Enable	`BOOL`
RS Ambient	`D3DCOLOR`
RS Ambient Material Source	`D3DMATERIALCOLORSOURCE`
RS Antialiased Line Enable	`BOOL`
RS Blend Factor	`float`
RS Blend Op	`D3DBLENDOP`
RS Blend Op Alpha	`D3DBLENDOP`
RS CCW Stencil Fail	`D3DSTENCILOP`
RS CCW Stencil Func	`D3DCMPFUNC`
RS CCW Stencil Pass	`D3DSTENCILOP`
RS CCW Stencil Z Fail	`D3DSTENCILOP`
RS Clipping	`BOOL`
RS Clip Plane Enable	`BOOL`
RS Color Write Enable	`D3DCOLORWRITEENABLE`
RS Color Write Enable $1-3$	`D3DCOLORWRITEENABLE`
RS Color Vertex	`BOOL`

. . . continued

Device State Data Types (continued)

Device State	Data Type
RS Cull Mode	D3DCULL
RS Debug Monitor Token	D3DDEBUGMONITORTOKENS
RS Depth Bias	float
RS Dest Blend	D3DBLEND
RS Dest Blend Alpha	D3DBLEND
RS Diffuse Material Source	D3DMATERIALCOLORSOURCE
RS Dither Enable	BOOL
RS Emissive Material Source	D3DMATERIALCOLORSOURCE
RS Enable Adaptive Tessellation	BOOL
RS Fill Mode	D3DFILLMODE
RS Fog Color	D3DCOLOR
RS Fog Density	float
RS Fog Enable	BOOL
RS Fog End	float
RS Fog Start	float
RS Fog Table Mode	D3DFOGMODE
RS Fog Vertex Mode	D3DFOGMODE
RS Indexed Vertex Blend Enable	BOOL
RS Last Pixel	BOOL
RS Lighting	BOOL
RS Local Viewer	BOOL
RS Max Tessellation Level	DWORD
RS Min Tessellation LEvel	DWORD
RS Normal Degree	D3DDEGREETYPE
RS Normalize Normals	BOOL
RS Multi Sample Antialias	BOOL
RS Multi Sample Mask	$[0, 2^m - 1]$
RS Normal Degree	D3DDEGREETYPE
RS Patch Edge Style	D3DPATCHEDGESTYLE
RS Point Scale A, B, C	float
RS Point Scale Enable	BOOL
RS Point Size	float
RS Point Size Max	float
RS Point Size Min	float
RS Point Sprite Enable	BOOL
RS Position Degree	D3DDEGREETYPE
RS Range Fog Enable	BOOL
RS Scissor Test Enable	BOOL
RS Separate Alpha Blend Enable	BOOL
RS Shade Mode	D3DSHADEMODE
RS Slope Scale Depth Bias	float
RS Specular Enable	BOOL
RS Specular Material Source	D3DMATERIALCOLORSOURCE

. . . continued

Device State Data Types (continued)

Device State	Data Type
RS Src Blend	D3DBLEND
RS Src Blend Alpha	D3DBLEND
RS sRGB Write Enable	BOOL
RS Stencil Enable	BOOL
RS Stencil Fail	D3DSTENCILOP
RS Stencil Func	D3DCMPFUNC
RS Stencil Mask	$[0, 2^s - 1]$
RS Stencil Pass	D3DSTENCILOP
RS Stencil Ref	$[0, 2^s - 1]$
RS Stencil Write Mask	$[0, 2^s - 1]$
RS Stencil Z Fail	D3DSTENCILOP
RS Texture Factor	D3DCOLOR
RS Tween Factor	float
RS Two Sided Stencil Mode	BOOL
RS Vertex Blend	D3DVERTEXBLENDFLAGS
RS Wrap $0 - 15$	D3DWRAPCOORD
RS Z Enable	D3DZBUFFERTYPE
RS Z Func	D3DCMPFUNC
RS Z Write Enable	BOOL
SS Address U, V, W	D3DTEXTUREADDRESS
SS Border Color	D3DCOLOR
SS D Map Offset	DWORD
SS Element Index	DWORD
SS Mag Filter	D3DTEXTUREFILTERTYPE
SS Max Anisotropy	DWORD
SS Max Mip Level	$[0, l - 1]$
SS Min Filter	D3DTEXTUREFILTERTYPE
SS Mip Filter	D3DTEXTUREFILTERTYPE
SS Mip Map LOD Bias	float
SS sRGB Texture	BOOL
TSS Alpha Arg $0 - 2$	D3DTEXTUREARG
TSS Alpha Op	D3DTEXTUREOP
TSS Bump Env L Offset	float
TSS Bump Env L Scale	float
TSS Bump Env Mat $00 - 11$	float
TSS Color Arg $0 - 2$	D3DTEXTUREARG
TSS Color Op	D3DTEXTUREOP
TSS Constant	float
TSS Result Arg	D3DTEXTUREARG
TSS Tex Coord Index	$[0, 15]$ | D3DTSS_TCI
TSS Texture Transform Flags	D3DTEXTURETRANSFORMFLAGS

Table 3.6: Device State Data Types. $[i, j]$ indicates a DWORD value restricted to the given interval. s is the number of bits in the stencil buffer. m is the number of multisamples. l is the number of texture levels. D3DTSS_TEXCOORDINDEX has an index in the LOWORD and D3DTSS_TCI flags in the HIWORD. [1]BOOL is a type alias for an unsigned integer used with the constants TRUE and FALSE.

3.9.1 Miscellaneous Render States

RS Debug Monitor Token does not control rendering and is not associated with any particular portion of the pipeline. RS Debug Monitor Token controls debug information returned by the pipeline as a whole when using the debug runtime. See chapter 22 for more information on the debug runtime. The debug tokens are defined by the `D3DDEBUGMONITORTOKENS` enumeration. The currently defined debug monitor tokens enable or disable the reporting of debug information from the runtime.

```
typedef enum _D3DDEBUGMONITORTOKENS {
    D3DDMT_ENABLE  = 0,
    D3DDMT_DISABLE = 1
} D3DDEBUGMONITORTOKENS;
```

3.9.2 Setting Groups of State

Often in a Direct3D application we have a collection of render and texture stage states that need to be set. Writing out each value to be set as a call to `SetRenderState` or `SetTextureStageState` is tedious. We can automate this with the following code idiom contained in the file `<rt/states.h>` located in the sample code.

Listing 3.1: `<rt/states.h>`: Setting aggregate state

```
1   #if !defined(RT_STATES_H)
2   #define RT_STATES_H
3   //-------------------------------------------------------------
4   // states.h
5   //
6   // Helper functions for setting render and texture stage states.
7   //
8   #include <d3d9.h>
9
10  namespace rt
11  {
12
13  //-------------------------------------------------------------
14  // s_enum_value<Enum>
15  // s_rs
16  // s_tss
17  //
18  // struct for storing a device state and its value
19  template <typename Enum>
20  struct s_enum_value
21  {
22      Enum m_state;
```

```
23      DWORD m_value;
24  };
25  typedef s_enum_value<D3DRENDERSTATETYPE> s_rs;
26  typedef s_enum_value<D3DTEXTURESTAGESTATETYPE> s_tss;
27  typedef s_enum_value<D3DSAMPLERSTATETYPE> s_ss;
28
29  //-----------------------------------------------------------
30  // set_states
31  //
32  // Set render states on the device.
33  //
34  inline void
35  set_states(IDirect3DDevice9 *device,
36             const s_rs *states, UINT num_states)
37  {
38      for (UINT i = 0; i < num_states; i++)
39      {
40          THR(device->SetRenderState(states[i].m_state,
41              states[i].m_value));
42      }
43  }
44
45  //-----------------------------------------------------------
46  // set_states
47  //
48  // Set texture stage states on the device.
49  //
50  inline void
51  set_states(IDirect3DDevice9 *device, UINT stage,
52             const s_tss *states, UINT num_states)
53  {
54      for (UINT i = 0; i < num_states; i++)
55      {
56          THR(device->SetTextureStageState(stage,
57              states[i].m_state, states[i].m_value));
58      }
59  }
60
61  inline void
62  set_states(IDirect3DDevice9 *device, UINT stage,
63             const s_ss *states, UINT num_states)
64  {
65      for (UINT i = 0; i < num_states; i++)
66      {
67          THR(device->SetSamplerState(stage,
68              states[i].m_state, states[i].m_value));
```

```
69          }
70      }
71
72      //-------------------------------------------------------------
73      // float_dword, dword_float
74      //
75      // Reinterpret a float as a DWORD and vice-versa for device
76      // states represented as a float crammed into a DWORD.
77      //
78      inline DWORD
79      float_dword(float value)
80      {
81          return *reinterpret_cast<DWORD *>(&value);
82      }
83      inline float
84      dword_float(DWORD value)
85      {
86          return *reinterpret_cast<float *>(&value);
87      }
88
89      };  // rt
90
91      #endif
```

The following code illustrates the use of `set_states` for render and texture stage states.

```
// number of elements in a fixed-size array
#define NUM_OF(ary_) (sizeof(ary_)/sizeof(ary_[0]))

// render states
const rt::s_rs render_states[] =
{
    D3DRS_LIGHTING, FALSE,
    D3DRS_COLORVERTEX, TRUE,
    D3DRS_SPECULARENABLE, FALSE,
    D3DRS_ALPHATESTENABLE, FALSE,
    D3DRS_ZENABLE, D3DZB_FALSE,
    D3DRS_DITHERENABLE, TRUE
};
rt::set_states(device, render_states, NUM_OF(render_states));

// stage 0 texture stage states.
const rt::s_tss texture0_states[] =
{
```

```
    D3DTSS_COLORARG1, D3DTA_TEXTURE,
    D3DTSS_COLOROP, D3DTOP_MODULATE,
    D3DTSS_COLORARG2, D3DTA_DIFFUSE,
    D3DTSS_ALPHAARG1, D3DTA_TEXTURE,
    D3DTSS_ALPHAOP, D3DTOP_SELECTARG1
};
rt::set_states(device, 0,
    texture0_states, NUM_OF(texture0_states));

// stage 0 sampler states
const rt::s_ss sampler_sates[] =
{
    D3DTSS_ADDRESSU, D3DTADDRESS_CLAMP,
    D3DTSS_ADDRESSV, D3DTADDRESS_CLAMP,
    D3DTSS_MINFILTER, D3DTEXF_LINEAR,
    D3DTSS_MAGFILTER, D3DTEXF_LINEAR,
    D3DTSS_MIPFILTER, D3DTEXF_NONE
};
rt::set_states(device, 0,
    sampler_states, NUM_OF(sampler_states));

// stage 1 texture stage states
const rt::s_tss texture1_states[] =
{
    D3DTSS_COLOROP, D3DTOP_DISABLE,
    D3DTSS_ALPHAOP, D3DTOP_DISABLE
};
rt::set_states(device, 1,
    texture1_states, NUM_OF(texture1_states));
```

`ID3DXEffect` and device state blocks provide alternative ways of managing collections of device state. `ID3DXEffect` is described in section 18.2. Device state blocks are described in the next section.

3.10 Device State Blocks

State blocks are COM objects that provide a way for your application to cache a group of device properties for later use. For instance, two state blocks could be used for the device properties corresponding to the unselected and selected appearance of an object in an editor. The `IDirect3DStateBlock9` interface is summarized in interface 3.4.

Each state block is associated with a device, returned by the `GetDevice` method. Once a state block has been created, the state block can be applied to the device by calling `Apply` on the state block. Calling `Capture` on an existing state block captures the current values of the device properties into the state block.

Interface 3.4: Summary of the `IDirect3DStateBlock9` interface.

IDirect3DStateBlock9

Read-Only Properties	
`GetDevice`	The associated device.
Methods	
`Apply`	Applies the block's state to the device.
`Capture`	Captures the current device state into the block.

```
interface IDirect3DStateBlock9 : IUnknown
{
 // read-only properties
 HRESULT GetDevice(IDirect3DDevice9 **value);

 // methods
 HRESULT Apply();
 HRESULT Capture();
};
```

There are two ways to create a state block object and fill it with specific device property values. The first way to create a state block is to call `CreateStateBlock` with a `D3DSTATEBLOCKTYPE` value identifying the kind of state you want recorded in the block. Table 3.8 summarizes the device properties included in the state block created with `CreateStateBlock`.

```
HRESULT CreateStateBlock(D3DSTATEBLOCKTYPE kind,
            IDirect3DStateBlock9 **result);

typedef enum _D3DSTATEBLOCKTYPE
{
    D3DSBT_ALL           = 1,
    D3DSBT_PIXELSTATE    = 2,
    D3DSBT_VERTEXSTATE   = 3,
} D3DSTATEBLOCKTYPE;
```

The second way of obtaining a state block object is to call `BeginStateBlock`, set device properties and then call `EndStateBlock`. Once `BeginStateBlock` has been called, the methods listed in table 3.7 mark their state for capture.

```
HRESULT BeginStateBlock();
HRESULT EndStateBlock(IDirect3DStateBlock9 **result);
```

`LightEnable`	`SetSamplerState`
`SetClipPlane`	`SetStreamSource`
`SetIndices`	`SetTexture`
`SetLight`	`SetTextureStageState`
`SetMaterial`	`SetTransform`
`SetPixelShader`	`SetVertexShader`
`SetPixelShaderConstantB`	`SetVertexShaderConstantB`
`SetPixelShaderConstantF`	`SetVertexShaderConstantF`
`SetPixelShaderConstantI`	`SetVertexShaderConstantI`
`SetRenderState`	`SetViewport`

Table 3.7: Methods captured by `BeginStateBlock`.

When `EndStateBlock` is called, each device property marked for capture is recorded into the state block. If a device property is set multiple times between `BeginStateBlock` and `EndStateBlock`, only the last value set in the property is captured into the state block.

State blocks do not survive the loss of a device and will need to be destroyed and recreated when the device is regained. Release the state block COM object when you are finished with a state block.

Device Properties by State Block Type

Device Property	Vertex	Pixel	All
Clip Planes			X
Light	X		X
Light Enable			X
Material			X
Pixel Shader		X	X
Pixel Shader Constant		X	X
Texture			X
Texture Palette			X
Transform			X
Stream Source			X
Vertex Shader	X		X
Vertex Shader Constant	X		X
Viewport			X
RS Adaptive Tess X, Y, Z, W	X	X	
RS Alpha Blend Enable		X	X
RS Alpha Func		X	X
RS Alpha Ref		X	X
RS Ambient	X		X
RS Ambient Material Source	X		X
RS Antialiased Line Enable		X	X
RS Blend Factor		X	X
RS Blend Op		X	X

. . . continued

Device Properties by State Block Type (continued)

Device Property	Vertex	Pixel	All
RS Blend Op Alpha		X	X
RS CCW Stencil Fail		X	X
RS CCW Stencil Func		X	X
RS CCW Stencil Pass		X	X
RS CCW Stencil Z Fail		X	X
RS Clipping	X		X
RS Clip Plane Enable	X		X
RS Color Vertex	X		X
RS Color Write Enable		X	X
RS Color Write Enable $1-3$		X	X
RS Depth Bias	X		X
RS Dest Blend		X	X
RS Dest Blend Alpha		X	X
RS Diffuse Material Source	X		X
RS Dither Enable		X	X
RS Emissive Material Source	X		X
RS Enable Adaptive Tessellation	X	X	
RS Fill Mode		X	X
RS Fog Density	X	X	X
RS Fog End	X	X	X
RS Fog Start	X	X	X
RS Fog Table Mode	X		X
RS Fog Vertex Mode	X		X
RS Indexed Vertex Blend Enable	X		X
RS Last Pixel		X	X
RS Lighting	X		X
RS Local Viewer	X		X
RS Max Tessellation Level	X	X	
RS Min Tessellation LEvel	X	X	
RS Multi Sample Antialias	X		X
RS Multi Sample Mask	X		X
RS Normalize Normals	X		X
RS Patch Edge Style	X		X
RS Point Scale A, B, C	X		X
RS Point Scale Enable	X		X
RS Point Size	X		X
RS Point Size Max	X		X
RS Point Size Min	X		X
RS Point Sprite Enable	X		X
RS Range Fog Enable	X		X
RS Scissor Test Enable	X		X
RS Separate Alpha Blend Enable	X		X
RS Shade Mode		X	X

. . . continued

Device Properties by State Block Type (continued)

Device Property	Vertex	Pixel	All
RS Slope Scale Depth Bias	X	X	
RS Specular Material Source	X		X
RS Src Blend		X	X
RS Src Blend Alpha		X	X
RS sRGB Write Enable		X	X
RS Stencil Enable		X	X
RS Stencil Fail		X	X
RS Stencil Func		X	X
RS Stencil Mask		X	X
RS Stencil Pass		X	X
RS Stencil Ref		X	X
RS Stencil Write Mask		X	X
RS Stencil Z Fail		X	X
RS Texture Factor		X	X
RS Tween Factor	X		X
RS Two Sided Stencil Mode		X	X
RS Vertex Blend	X		X
RS Wrap $0-15$		X	X
RS Z Enable		X	X
RS Z Func		X	X
RS Z Write Enable		X	X
SS Address U,V, W		X	X
SS Border Color		X	X
SS D Map Offset		X	X
SS Element Index		X	X
SS Mag Filter		X	X
SS Max Anisotropy		X	X
SS Max Mip Level		X	X
SS Min Filter		X	X
SS Mip Filter		X	X
SS Mip Map LOD Bias		X	X
SS sRGB Texture		X	X
TSS Alpha Arg $0-2$		X	X
TSS Alpha Op		X	X
TSS Bump Env L Offset		X	X
TSS Bump Env L Scale		X	X
TSS Bump Env Mat $00-11$		X	X
TSS Color Arg $0-2$		X	X
TSS Color Op		X	X
TSS Constant		X	X
TSS Result Arg		X	X
TSS Tex Coord Index	X	X	X
TSS Texture Transform Flags	X	X	X

. . . continued

Device Properties by State Block Type (continued)			
Device Property	Vertex	Pixel	All

Table 3.8: Device Properties by State Block Type

3.11 Pure Devices

When creating the device, we saw that we could request a "pure" hardware device that performed minimal device state management. Table 3.9 summarizes the device properties available on a pure device. A pure device has a performance advantage because the runtime and driver do not have to keep a copy of the non-queryable state for the application. If the `D3DDEVCAPS_PUREDEVICE` bit is set in `D3DCAPS9::DevCaps`, then the driver supports a pure device.

```
#define D3DDEVCAPS_PUREDEVICE 0x00100000L
```

Supported	Not Supported
GetAvailableTextureMem	GetClipPlane
GetBackBuffer	GetClipStatus
GetCreationParameters	GetLight
GetCurrentTexturePalette	GetLightEnable
GetDepthStencilSurface	GetMaterial
GetDeviceCaps	GetPaletteEntries
GetDirect3D	GetPixelShaderConstantB
GetDisplayMode	GetPixelShaderConstantF
GetFrontBufferData	GetPixelShaderConstantI
GetFVF	GetRenderState
GetGammaRamp	GetSamplerState
GetIndices	GetTextureStageState
GetNPatchMode	GetTransform
GetNumberOfSwapChains	GetVertexShaderConstantB
GetPixelShader	GetVertexShaderConstantF
GetRasterStatus	GetVertexShaderConstantI
GetRenderTarget	SetClipStatus
GetRenderTargetData	
GetScissorRect	
GetSoftwareVertexProcessing	
GetStreamSource	
GetStreamSourceFreq	
GetSwapChain	
GetTexture	
GetVertexDeclaration	
GetVertexShader	
GetViewport	

Table 3.9: Device property support on a pure HAL device.