Chapter 5

Modeling

"Memory is the treasury and guardian of all things."
Cicero: *De orature*, I, c. 80 B.C.

5.1 Overview

Direct3D produces pixel images from scenes by processing geometry and appearance data into pixels through a process called rasterization. In this chapter, we introduce the mechanism of describing scenes to Direct3D. Scenes contain one or more objects that are positioned relative to each other and to the virtual camera that views the scene. Such objects are called "models", and contain data that define their shape and appearance. The shape of a model is described by a collection of simple geometric shapes, called graphic primitives. The appearance of a model is described by providing data associated with each vertex.

When we draw a three dimensional scene, we need to solve the problem of visibility: objects in the foreground should occlude objects in the background. Two dimensional applications use a painter's algorithm to determine visibility, but this is a problematic approach for complex three dimensional scenes. Direct-3D can use depth/stencil surfaces to resolve visibility.

Models in the scene are made from a collection of primitive shapes to form the surface of the entire model. Direct3D only provides primitives for modeling the surface of an object, leading to scenes composed of object shells. Direct3D provides primitive shapes for points, lines and triangles. Since triangles are planar, all smooth surfaces are approximated with a collection of triangles. Additional information about the true surface normal at each vertex of the approximation can be included to improve the object's appearance, see chapter 8.

Additional data besides position and surface normal can be given at each vertex. Using either a flexible vertex format or a vertex shader declaration, Direct-3D describes colors, texture coordinates, vertex blending weights and arbitrary shader data at each vertex. The vertex data is supplied to the device through

a collection of streams, each associated with a vertex buffer. The streams can be driven through a level of indirection with an index buffer.

Direct3D also provides two ways to describe a shape parametrically, with the parameters defining a refinement of a basic surface description into an arbitrary collection of triangles approximating the true surface.

The scene is described to the device one primitive at a time. Each primitive is rasterized into a collection of pixels written to the render target property of the device. We describe how to manipulate the render target property and how the render target interacts with the back buffers of a swap chain.

Not every object we want to model is described exactly by a geometric primitive. An application can use its own software rendering and direct pixel manipulation, as in chapter 4, but usually there is a way to model the desired object with a suitable collection of graphic primitives. We describe a few examples of objects such as drawing text, non-rectangular planar objects, volumetric objects and level of detail approximations.

5.2 Modeling Scenes

All Direct3D rendering is done within a scene. `BeginScene` marks the beginning of a scene and precedes all graphic rendering. `EndScene` marks the end of a scene and follows all graphic rendering. Marking the scene in this manner allows devices to perform computations that should be executed once for each frame of rendering. Calling `EndScene` when not in a scene, `BeginScene` while in a scene, or performing any rendering outside of a scene all return `D3DERR_INVALIDCALL`.

```
HRESULT BeginScene();
HRESULT EndScene();
```

Device properties and states can be set at any time after the device has been created, they are not affected by scenes. `StretchRect` is not considered a rendering method and can be used outside of a scene. `StretchRect` is often implemented using a separate data path than that used for rendering. If possible, avoid interleaving calls to `StretchRect` with rendering calls. See chapter 23 for more on performance issues related to the use of the pipeline.

Typically an application will initialize the device to a known state once the device has been created and then call `BeginScene` and `EndScene` around an internal traversal of a hierarchical database representing the visible scene. Direct3D provides no direct facilities for the scene database, which is usually tightly coupled to the application's internal data structures. At the scene level, D3DX provides mesh objects for representing models as a whole, see chapter 19 for more on mesh objects. D3DX also provides a matrix stack object that is useful in traversing a scene hierarchy. See chapter 17 for more on the matrix stack object.

Animating a scene involves rendering a sequence of frames, each bracketed by `BeginScene` and `EndScene`. Each successive frame draws the models at successive moments in time, as in cel animation used for cartoons. If the frames

can be drawn rapidly enough, the eye will perceive the sequence of images as a continuously moving image. Projected motion pictures present frames at a rate of 24 each second. Direct3D can present frames as fast as the refresh rate of the monitor for a synchronized presentation, or faster with immediate presentation. Frame rates of greater than 15 fps generally are perceived as "real-time" and can provide the sense of immersion in a simulation. Techniques used in "real-time" simulation build on the basic features of Direct3D and are too lengthy to describe here, see section 5.18.

The objects in scenes can be created programmatically, that is by writing a program to compute their shape and appearance. The more common case is to create models in an application specifically designed for editing models, such as *Maya*, *3D Studio Max* and *Softimage*. Models are written into a file from the editing application and read from a file by the rendering application when creating resources. An application can use the modeler's file format, its own file format, or the X file format. Using the format of the modeling package may be convenient, provided documentation or a parsing library for the file format exists. Creating your own file format from scratch is tedious and time consuming work. Writing the necessary file manipulation routines for reading and writing scenes and models is error-prone. (You will also need to write your own conversion tool to and from the modeling package's file format.)

An application can use the X file format provided with the SDK instead of inventing another file format for scene and model data. The X file format is extensible and easily adaptable to the specific needs of an application and provides basic model storage immediately. See chapter 21 for more information about X file support in the SDK. The SDK also provides plugins for some common modeling packages that allow them to use the X file format for models, see appendix A.

5.3 Visibility

Visibility isn't much of a problem in two dimensional applications such as those that use GDI. They view the window's client area as a place where they draw a stack of objects from "bottom" to "top", in that order. The topmost object is drawn last and covers over anything that was drawn under its pixel locations. This is called the "painter's algorithm" solution to the visibility problem. It works fine when the objects are all planar and don't intersect as they would in a GDI application.

However, the natural extension of the painter's algorithm to three dimensions fails in the general case. If we sort the models in the scene from back to front and draw the models in that order, this is the equivalent of painter's algorithm in three dimensions. As long as the models don't interpenetrate or form a cycle of occlusion, the algorithm works fine. A cycle of occlusion can be visualized with a scene containing four overlapping rectangles, such as in figure 5.1. Rectangle A occludes B, B occludes C, C occludes D, and D occludes A, giving rise to a cycle of occlusion. While any two of the rectangles have a well defined ordering

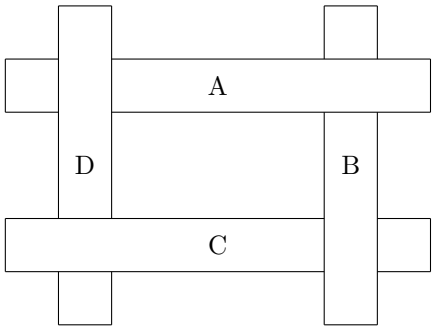


Figure 5.1: Illustration of a cycle of occlusion with four rectangles. This scene cannot be drawn properly by the standard painter's algorithm for resolving visibility.

that determines which rectangle is in front, taken as a whole no one rectangle can be considered to be in front of all the others.

The "Z-buffer algorithm" solution to the visibility problem works at each pixel on the render target instead of on models as a whole. As models are rasterized, each pixel will have an associated Z value that is used to determine which pixel is closest to the camera. The closer pixels are stored into the render target, while pixels farther away are discarded.

While the Z-buffer algorithm is superior to painter's algorithm it is not perfect. In particular, the Z-buffer assumes that pixels are completely opaque and the pixel nearest the camera is always "on top." However, if the model is partially transparent, then its pixels will be partially transparent and whatever is "behind" those partially transparent pixels should be seen. Fortunately, acceptable results can usually be obtained by drawing all solid models first and then roughly sort the transparent objects from back to front and draw them with painter's algorithm. This approach is not perfect, but it does reduce the artifacts while still exploiting the fast Z-buffer hardware.

The depth/stencil buffer holds each pixel's depth from the camera. A depth/stencil buffer can be created as a property of the device based on the `D3DPRESENT_PARAMETERS` for the device, or explicitly by calling `CreateDepthStencilSurface`. The depth/stencil buffer surface created with the device is returned by `GetDepthStencilSurface`. To use the Z-buffer for resolving visibility, set RS Z Enable to `D3DZB_TRUE`, RS Z Write Enable to `TRUE`, and RS Z Func to `D3DCMP_LESS`. The operation of the depth/stencil buffer is described in detail in chapter 14.

If the `D3DPRASTERCAPS_ZBUFFERLESSHSR` bit of `D3DCAPS9::RasterCaps` is set, it indicates the device has an alternative visibility algorithm for hidden surface removal (HSR) that doesn't use a Z-buffer. How visibility is determined is hardware dependent and application transparent. Typically this capability is exposed by so-called "scene capture" cards. In this case, visibility can be resolved by setting the depth/stencil buffer for the render target to `NULL`, and

setting RS Z Enable to D3DZB_TRUE.

```
#define D3DPRASTERCAPS_ZBUFFERLESSHSR 0x00008000L
```

5.4 Render Targets

As primitives are rasterized, a stream of pixels is produced. The destination for this stream of pixels is the render target property on the device. All back buffers on swap chains are valid render targets, but render targets are not restricted to back buffer surfaces. When the device is created or reset, the render target is back buffer zero of the device's default swap chain. When `Present` is called on the device, the render target advances to the next back buffer so that after `Present` returns, render target is back buffer zero again. This means that an application which only renders to the back buffer and presents frames never needs to explicitly set the render target property.

Setting the render target to a surface other than a back buffer surface allows the device to render directly into an image surface instead of rendering into a swap chain's back buffer and using `StretchRect` to obtain the results of the rendering, which could stall the pipeline. It also allows a device to render directly into a surface contained in a cube map or in a texture map as we will see in chapter 11.

A render target surface not associated with a swap chain can be created by calling `CreateRenderTarget`. The format of the render target must be validated with `CheckDeviceFormat` for `D3DRTYPE_SURFACE` with `D3DUSAGE_RENDERTARGET`. If the `sampling` argument is not `D3DMULTISAMPLE_NONE`, then it must be validated with `CheckDeviceMultiSampleType` for the render target surface format.

```
HRESULT CreateRenderTarget(UINT width,
            UINT height,
            D3DFORMAT format,
            D3DMULTISAMPLE_TYPE sampling,
            BOOL lockable,
            IDirect3DSurface9 **result);
HRESULT GetRenderTarget(IDirect3DSurface9 **value);
HRESULT SetRenderTarget(IDirect3DSurface9 *value,
            IDirect3DSurface9 *depth_stencil);
```

`GetRenderTarget` returns a surface interface for the render target property of the device. The render target can be changed with `SetRenderTarget`, which takes two surface interfaces for the color buffer and depth/stencil buffer of the new render target. The color buffer surface and depth/stencil buffer surface must match in pixel dimensions and their respective formats must be validated with `CheckDepthStencilMatch`. If the `depth_stencil` argument `NULL`, then no depth/stencil buffer is associated with the new render target. If the `value` parameter is `NULL`, then the existing render target's color buffer is retained.

The `Clear` method initializes the render target to known values. The `color`, `z`, and `stencil` arguments provide the values to be stored in the render target. The `flags` argument controls which values are written and must be one or more of the following flags.

```
HRESULT Clear(DWORD num_rects,
              const D3DRECT *rects,
              DWORD flags,
              D3DCOLOR color,
              float z,
              DWORD stencil);

#define D3DCLEAR_TARGET  0x00000001L
#define D3DCLEAR_ZBUFFER 0x00000002L
#define D3DCLEAR_STENCIL 0x00000004L
```

The `num_rects` and `rects` arguments restrict the clear operation to a subregion of the viewport into the render target surface. If `rects` is `NULL`, then the entire viewport is cleared. Unless changed, the viewport always covers the entire render target surface and is reset to the entire render target surface when the render target is changed. The viewport property restricts rendering to a portion of the render target surface and is described in section 6.11.

5.5 Primitive Types

Models in the scene are made from a collection of primitive shapes to form the surface of the entire model. Direct3D only provides primitives for modeling the surface of an object, leading to scenes composed of object shells. Direct3D provides primitive shapes for points, lines, triangles and higher-order surface patches. Since triangles are planar, they can only approximate true smooth surfaces. Additional information about the true surface normal at each vertex of the approximation can be included to provide a more realistic appearance to the object. Direct3D supports the primitive types given by the `D3DPRIMITIVETYPE` enumeration and illustrated in figure 5.2.

```
typedef enum _D3DPRIMITIVETYPE {
    D3DPT_POINTLIST     = 1,
    D3DPT_LINELIST      = 2,
    D3DPT_LINESTRIP     = 3,
    D3DPT_TRIANGLELIST  = 4,
    D3DPT_TRIANGLESTRIP = 5,
    D3DPT_TRIANGLEFAN   = 6,
} D3DPRIMITIVETYPE;
```

`D3DPT_POINTLIST` draws a collection of points, the vertex labels in the figure are for reference only and are not drawn by Direct3D. `D3DPT_LINELIST` draws

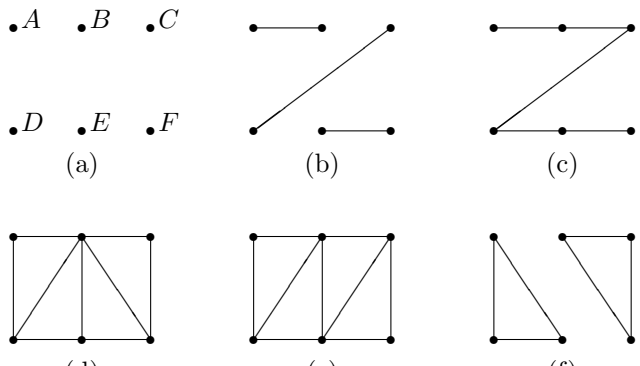


Figure 5.2: Illustration of primitive types with the vertices {A, B, C, D, E, F}, unless otherwise noted. (a) `D3DPT_POINTLIST`, (b) `D3DPT_LINELIST`, (c) `D3DPT_LINESTRIP`, (d) `D3DPT_TRIANGLEFAN` {B, C, F, E, D, A}, (e) `D3DPT_TRIANGLESTRIP` {D, A, B, E, C, F}, (f) `D3DPT_TRIANGLELIST` {A, E, D, B, C, F}.

a sequence of possibly disjoint line segments and `D3DPT_LINELIST` draws a sequence of connected line segments.

Solid shapes are drawn with collections of triangles. `D3DPT_TRIANGLEFAN` draws a sequence of triangles where each triangle after the first shares an edge with the previous triangle. In figure 5.2(d), the triangles generated are BCF, BFE, BED, and BDA. Triangles BCF and BFE share edge BF.

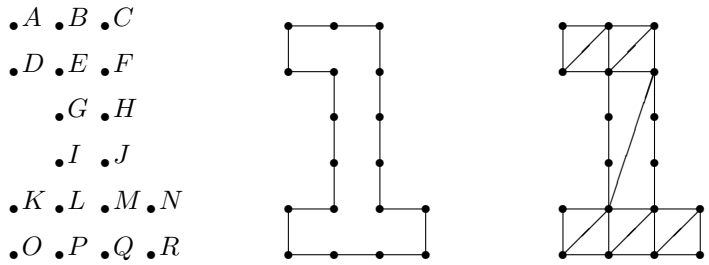
`D3DPT_TRIANGLESTRIP` draws a strip of triangles, each sharing a common edge as well, but in a different topology from `D3DPT_TRIANGLEFAN`. In figure 5.2(e), the triangles generated are DAB, DBE, EBC, and ECF. Here each triangle after the first shares a common edge with the previous triangle. Triangles DAB and DBE share edge DB.

`D3DPT_TRIANGLELIST` draws a sequence of possibly disjoint triangles. In figure 5.2(f), the triangles generated are AED and BCF. The triangles needn't share any vertices or edges, making this the most general primitive type for triangles.

Figure 5.3 shows three ways to draw the shape of the numeral 1. The points in figure 5.3(a) can be drawn with `D3DPT_POINTLIST`, and the order of the vertices is not significant. The point labels are not part of the primitive, they are shown only for reference. The vertex indices will be discussed in section 5.10.

The outline in (b) can be drawn with `D3DPT_LINESTRIP` and the vertices {A, C, M, N, R, O, K, L, E, A}. The first vertex must be repeated at the end to close the loop. The outline could also be drawn as a series of disconnected segments using `D3DPT_LINELIST` and the vertices {A, C, C, M, M, N, N, R, R, O, O, K, K, L, L, E, E, A}.

The triangulation in (c) can be drawn with 3 `D3DPT_TRIANGLESTRIP` primitives with the vertices {A, D, B, E, C, F}, {E, L, F, M}, and {K, O, L, P, M, Q, N, R}. It can be drawn with a single `D3DPT_TRIANGLELIST` with the vertices {A, B, D, B, E, D, B, C, E, C, F, E, E, F, L, F, M, L, K, L,



(a) (b) (c)

Point	Index	Position	Point	Index	Position	Point	Index	Position
A	0	$(0, 5)$	G	6	$(1, 3)$	M	12	$(2, 1)$
B	1	$(1, 5)$	H	7	$(2, 3)$	N	13	$(3, 1)$
C	2	$(2, 5)$	I	8	$(1, 2)$	O	14	$(0, 0)$
D	3	$(0, 4)$	J	9	$(2, 2)$	P	15	$(1, 0)$
E	4	$(1, 4)$	K	10	$(0, 1)$	Q	16	$(2, 0)$
F	5	$(3, 4)$	L	11	$(1, 1)$	R	17	$(3, 0)$

(d)

Figure 5.3: Modeling with primitive types. (a) points defining the shape of the numeral 1, (b) outlined, (c) triangulated, (d) vertex data

O, L, P, O, L, M, P, M, Q, P, M, N, Q, N, R, Q}. It can be drawn using 6 `D3DPT_TRIANGLEFAN` primitives with the vertices {B, E, D, A}, {C, F, E, B}, {F, M, L, E}, {L, P, O, K}, {M, Q, P, L}, and {N, R, Q, M}. These triangulations are only illustrative and are not the only triangulations of this shape.

When primitives share edges but not vertices, they may be mathematically coincident, but the finite accuracy in computer arithmetic gives rise to roundoff error in the coordinates as they are processed by Direct3D. If primitives share edges but not vertices, this can result in "cracks" appearing in the surface being modeled. The objects behind the surface show through the crack between primitive triangles. If vertex coordinates are shared, then the same roundoff error is applied to both primitives and cracks are avoided. Returning to figure 5.3, if triangles KNO and FML were drawn, cracks may apear along the segment LM in common between the two triangles, where they share no vertices. The points L and M are referred to as "T-junctions" and can be avoided by inserting additional vertices at points L and M, as in the triangulations of the preceding paragraph.

The best way to avoid T-junctions and the resulting cracks is to ensure that your modeling process and tools do not generate models with T-junctions. This is usually not a problem with commercial modeling packages. If you are programmatically triangulating shapes, you should ensure that your generated triangulation doesn't introduce T-junctions into the model.

Triangles are primitives that model the surface of an object, but not its interior. The triangle surface can be thought of as the dividing line between the space "inside" the model and the space "outside" the model. For a triangle, the outside direction is defined by its surface normal. The surface normal can be computed as the cross-product of the vectors formed from any two sides of the triangle. This means that if the triangle is in the plane of the page, listing its vertices in clockwise order gives a surface normal that points out of the page towards the reader in a left-handed coordinate system. If the vertices are listed in counter-clockwise order, then the surface normal will point into the page away from the reader.

A typical model's shape is an enclosed surface, such as a sphere. If the object contains a hole, such as a bottle, the interior surface of the object should also be modeled. If the interior surface is not modeled, the exterior surface polygons will be seen from the "inside" of their surface. Their surface normals point in the wrong direction for the interior surface. This will cause errors in vertex processing and lighting, resulting in an incorrect rendering.

5.6 Vertex Data

Scene data consists of data defining shape and data defining appearance. We've seen how the shape can be specified by primitives and the coordinates of their vertices. Additional data can be associated with each vertex to define appearance information that is used when the primitive is rasterized to determine the color of the rasterized pixels.

Data for use with the **fixed-function vertex processing** pipeline is described by either a flexible vertex format (FVF) code, or a vertex shader declaration. In chapter 9 we describe vertex shader programs can use arbitrary data at each vertex. Each piece of data associated with the vertex is called a **vertex component**. With an FVF, all the vertex data must be in a single stream, but a vertex shader declaration can divide the components into multiple streams.

In addition to choosing fixed-function vertex processing or programmable vertex processing, a program can supply fully processed vertices directly to the device, skipping vertex processing altogether. This approach stems from a time when transformation and lighting were expensive operations and hardware accelerators for them were not common. Hardware transformation and lighting are becoming commonplace, but fully transformed vertices are still useful for screen-space primitives, such as those drawn by `minimal.cpp` in listing 2.1.

5.7 Flexible Vertex Formats

Direct3D can describe a vertex with a flexible vertex format code or a vertex declaration. The FVF of a vertex is a `DWORD` containing one or more flags that describe the vertex components present in memory. The FVF flags and their corresponding vertex components are summarized in table 5.1. The vertex

components are laid out in memory in the same order as the vertex components listed in the table, starting with the position component.

```
#define D3DFVF_XYZ              0x002
#define D3DFVF_XYZRHW           0x004
#define D3DFVF_XYZB1            0x006
#define D3DFVF_XYZB2            0x008
#define D3DFVF_XYZB3            0x00a
#define D3DFVF_XYZB4            0x00c
#define D3DFVF_XYZB5            0x00e
#define D3DFVF_NORMAL           0x010
#define D3DFVF_PSIZE            0x020
#define D3DFVF_DIFFUSE          0x040
#define D3DFVF_SPECULAR         0x080
#define D3DFVF_TEX0             0x000
#define D3DFVF_TEX1             0x100
#define D3DFVF_TEX2             0x200
#define D3DFVF_TEX3             0x300
#define D3DFVF_TEX4             0x400
#define D3DFVF_TEX5             0x500
#define D3DFVF_TEX6             0x600
#define D3DFVF_TEX7             0x700
#define D3DFVF_TEX8             0x800
#define D3DFVF_LASTBETA_UBYTE4 0x1000
```

Every FVF must include a position with one of the D3DFVF_XYZ, D3DFVF_XYZRHW, D3DFVF_XYZB1, D3DFVF_XYZB2, D3DFVF_XYZB3, D3DFVF_XYZB4, or D3DFVF_XYZB5. The remaining FVF flags are all optional. The β value blending weights, D3DFVF_XYZBn and D3DFVF_LASTBETA_UBYTE4 flags are discussed in section 6.7. The D3DFVF_XYZ flag specifies untransformed vertex positions in model space, while the D3DFVF_XYZRHW specifies transformed vertices with positions transformed into screen space. A vertex transformation maps vertices in one coordinate frame to another coordinate frame with a matrix and is described in detail in chapter 7. Transformed vertices essentially pass through the pipeline unchanged until the vertices are rasterized. Lines 47–59 of listing 2.1 uses transformed vertices to draw a triangle, which is why the vertices of the triangle have their coordinates scaled by the dimensions of the window's client area.

D3DFVF_NORMAL indicates the presence of a true surface normal. D3DFVF_PSIZE is used only with point sprite primitives and specifies the size of the point sprite, as discussed in section 5.15.1. The D3DFVF_DIFFUSE and D3DFVF_SPECULAR flags indicate the presence of the diffuse and specular colors of the model's surface at the vertex. Diffuse and specular colors are discussed in more detail in chapter 8. Each vertex can have up to eight sets of texture coordinates, and each texture coordinate set can have one, two, three, or four dimensions. The D3DFVF_TEXn flags set the number of texture coordinate sets and the D3DFVF_TEXCOORDSIZEn macros set the dimensionality of the texture coordinate set argument as shown in table 5.1.

`D3DCAPS9::FVFCaps` describes the FVF capabilities of the device. The bits given by `D3DFVFCAPS_TEXCOORDCOUNTMASK` describe the maximum number of texture coordinates that the device can simultaneously use from each vertex for fixed-function pixel processing. The `D3DFVFCAPS_DONOTSTRIPELEMENTS` bit indicates that the device can handle vertices with unused vertex components. If this flag is not set, then it indicates that including vertex components in the FVF that are not used in rendering will render more slowly than if the unused vertex components were stripped from the vertex. For instance, if texturing is disabled, the device will render faster if texture coordinates are removed from the vertices.

If the `D3DFVFCAPS_PSIZE` bit is set, then the device supports a point size vertex component in the FVF for fixed-function processing of vertices in model space. Transformed vertices always support the point size vertex component and programmable vertex processing with a vertex shader always supports the output of a point size.

The `D3DFVFCAPS_TEXCOORDCOUNTMASK` bits of `FVFCaps` indicate the number of simultaneous texture coordinates that can be used by the device. A vertex may contain more texture coordinates, but only the number of coordinates specified by `D3DFVFCAPS_TEXCOORDCOUNTMASK` can be used for rendering.

```
#define D3DFVFCAPS_TEXCOORDCOUNTMASK  0x0000ffffL
#define D3DFVFCAPS_DONOTSTRIPELEMENTS 0x00080000L
#define D3DFVFCAPS_PSIZE              0x00100000L
```

Flexible Vertex Format

FVF Bits	Vertex Component or *FVF Structure Field*		
	FVF Flag	Data Type	Values
0	*Reserved*		
1-3	Position		
	`D3DFVF_XYZ`	`float[3]`	(x, y, z)
	`D3DFVF_XYZRHW`	`float[4]`	$(x, y, z, 1/w)$
	`D3DFVF_XYZB1`	`float[4]`	$(x, y, z), \beta_0$
	`D3DFVF_XYZB2`	`float[5]`	$(x, y, z), \beta_0, \beta_1$
	`D3DFVF_XYZB3`	`float[6]`	$(x, y, z), \beta_0, \beta_1, \beta_2$
	`D3DFVF_XYZB4`	`float[7]`	$(x, y, z), \beta_0, \ldots, \beta_3$
	`D3DFVF_XYZB5`	`float[8]`	$(x, y, z), \beta_0, \ldots, \beta_4$
4	Normal		
	`D3DFVF_NORMAL`	`float[3]`	$\langle n_x, n_y, n_z \rangle$
5	Point Size		
	`D3DFVF_PSIZE`	`float`	s
6	Diffuse Color		
	`D3DFVF_DIFFUSE`	`D3DCOLOR`	C_d
7	Specular Color		
	`D3DFVF_SPECULAR`	`D3DCOLOR`	C_s

...continued

Flexible Vertex Format (continued)

8-11	*Texture Coordinate Set Count*		
	D3DFVF_TEX0		
	D3DFVF_TEX1		
	D3DFVF_TEX2		
	D3DFVF_TEX3		
	D3DFVF_TEX4		
	D3DFVF_TEX5		
	D3DFVF_TEX6		
	D3DFVF_TEX7		
	D3DFVF_TEX8		
12	*Last β UBYTE4*		
	D3DFVF_LASTBETA_UBYTE4		
13-15	*Reserved*		
16-17	Texture Coordinate Set 0		
	D3DFVF_TEXCOORDSIZE1(0)	float	(s)
	D3DFVF_TEXCOORDSIZE2(0)	float[2]	(s,t)
	D3DFVF_TEXCOORDSIZE3(0)	float[3]	(s,t,u)
	D3DFVF_TEXCOORDSIZE4(0)	float[4]	(s,t,u,v)
18-19	Texture Coordinate Set 1		
	D3DFVF_TEXCOORDSIZE1(1)	float	(s)
	D3DFVF_TEXCOORDSIZE2(1)	float[2]	(s,t)
	D3DFVF_TEXCOORDSIZE3(1)	float[3]	(s,t,u)
	D3DFVF_TEXCOORDSIZE4(1)	float[4]	(s,t,u,v)
⋮		⋮	
30-31	Texture Coordinate Set 7		
	D3DFVF_TEXCOORDSIZE1(7)	float	(s)
	D3DFVF_TEXCOORDSIZE2(7)	float[2]	(s,t)
	D3DFVF_TEXCOORDSIZE3(7)	float[3]	(s,t,u)
	D3DFVF_TEXCOORDSIZE4(7)	float[4]	(s,t,u,v)

Table 5.1: The flexible vertex format fields and corresponding vertex components. Some fields in the FVF do not correspond directly to a vertex component and are shown in italic.

The FVF only defines the layout of the vertex components in memory, it doesn't matter what data structure you use for the vertex components as long as the ordering and memory layout is consistent with the FVF used. A commonly used vertex structure contains a position, a surface normal vector and a diffuse surface color. Here are several ways of representing this vertex as a data structure with its associated FVF.

```
DWORD fvf = D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE;
struct s_vertex1
{
```

```
    float m_pos[3];
    float m_normal[3];
    D3DCOLOR m_diffuse;
};

struct s_vertex2
{
    float m_x, m_y, m_z;
    float m_nx, m_ny, m_nz;
    DWORD m_color;
};

struct s_vertex3
{
    D3DVECTOR m_pos;
    D3DVECTOR m_normal;
    D3DCOLOR m_color;

    static const DWORD s_FVF;
};
const DWORD s_vertex3::s_FVF =
    D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_DIFFUSE;
```

If you use a data structure that doesn't match the FVF properly, this can be a difficult bug to find. To avoid this consider using a template class with an FVF template argument, or declaring the FVF as a static member of a vertex structure—static members in a `struct` or `class` aren't stored in each instance and therefore don't violate the layout constraints of the FVF. Repeating the FVF flags throughout your source code is going to be a source of bugs as your vertex data evolves with your application and it is recommended that you localize the FVF flags to one place and reference symbolically elsewhere. The following examples all exhibit *incorrect FVF usage*.

```
DWORD fvf1 = D3DFVF_XYZ | D3DFVF_DIFFUSE;
struct s_vertex1
{
    float m_pos[3];
    // Error! no D3DFVF_NORMAL in fvf, or
    //        superfluous component
    float m_normal[3];
    D3DCOLOR m_diffuse;
};

DWORD fvf2 = D3DFVF_XYZ | D3DFVF_DIFFUSE;
struct s_vertex2
{
```

```
    // Error! components out of order
    D3DCOLOR m_diffuse;
    float m_pos[3];
};

DWORD fvf3 = D3DFVF_XYZ | D3DFVF_DIFFUSE;
struct s_vertex3
{
    float m_pos[3];
    // Error! missing component, or
    //        superfluous FVF flag
};
```

The FVF property of the device is used to declare vertices by an FVF code. The `GetFVF` and `SetFVF` methods are used to manipulate the property.

```
HRESULT GetFVF(DWORD *value);
HRESULT SetFVF(DWORD value);
```

5.8 Vertex Declarations

A vertex declaration generalizes the FVF concept of describing vertices. The vertex is split into one or more streams, with each stream containing one or more vertex components. The declaration describes the number of streams and vertex components within each stream. A vertex declaration can be used with fixed-function vertex processing or with programmable vertex processing. Programmable vertex processing with vertex shaders is discussed in chapter 9. Using a vertex declaration is the only way to exploit multiple streams and vertex tweening with fixed-function processing.

A vertex declaration is created from an array of `D3DVERTEXELEMENT9` structures. Each element in the array compltely describes one vertex component within the vertex. The order of the elements in the array has no bearing on the layout of the data in the streams. With a vertex declaration, vertex component data can appear in any order in memory with respect to other components.

```
typedef struct _D3DVERTEXELEMENT9
{
    WORD Stream;
    WORD Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

The last element in the array of vertex elements is always a special “marker” element provided by the `D3DDECL_END` macro. A vertex declaration can have at

most MAXD3DDECLLENGTH elements in its array, not including the marker element. The Offset member gives the offset in bytes from the beginning of the Stream for this vertex element.

```
#define D3DDECL_END       { 0xFF, 0, D3DDECLTYPE_UNUSED, 0, 0, 0 }
#define MAXD3DDECLLENGTH  64
```

The Type member gives the datatype of the element, such as 3 floats or a D3DCOLOR and is given by the D3DDECLTYPE enumeration. The types D3DDECLTYPE_UBYTE4N through D3DDECLTYPE_FLOAT16_4 are only available in vertex shader model 2.0 or later.

```
typedef enum _D3DDECLTYPE
{
    D3DDECLTYPE_FLOAT1    =  0,
    D3DDECLTYPE_FLOAT2    =  1,
    D3DDECLTYPE_FLOAT3    =  2,
    D3DDECLTYPE_FLOAT4    =  3,
    D3DDECLTYPE_D3DCOLOR  =  4,
    D3DDECLTYPE_UBYTE4    =  5,
    D3DDECLTYPE_SHORT2    =  6,
    D3DDECLTYPE_SHORT4    =  7,
    D3DDECLTYPE_UBYTE4N   =  8,
    D3DDECLTYPE_SHORT2N   =  9,
    D3DDECLTYPE_SHORT4N   = 10,
    D3DDECLTYPE_USHORT2N  = 11,
    D3DDECLTYPE_USHORT4N  = 12,
    D3DDECLTYPE_UDEC3     = 13,
    D3DDECLTYPE_DEC3N     = 14,
    D3DDECLTYPE_FLOAT16_2 = 15,
    D3DDECLTYPE_FLOAT16_4 = 16,
    D3DDECLTYPE_UNUSED    = 17
} D3DDECLTYPE;
```

Conceptually all data types are expanded to four valued vectors before being passed on to vertex processing. If the data type doesn't already contain four values, then a value of zero will be provided for the y and z components and a value of one will be provided for the w component. Data type names follow this convention: *prefix base count suffix*. The prefix is either empty or U to indicate an unsigned quantity in the vertex data. The base type name is one of byte, short, float, D3DCOLOR or "dec". The count indicates the number of base types present in the data and the suffix indicates whether or not the data is normalized at the time of expansion. The D3DCOLOR vertex declaration type is expanded into the vector $\langle r, g, b, a \rangle$.

For example, D3DDECLTYPE_FLOAT1 is a single 32-bit floating-point value that will be expanded to the vector $\langle value, 0, 0, 1 \rangle$. D3DDECLTYPE_UBYTE4N normalizes the values from the $[0, 255]$ range to the $[0, 1]$ range.

TODO: Figure out layout of "DEC" type.

The Method member is used during vertex tessellation and is given by the D3DDECLMETHOD enumeration. Use D3DDECLMETHOD_DEFAULT when tessellation is not used; the remaining values are discussed in section 5.16.

```
typedef enum _D3DDECLMETHOD
{
    D3DDECLMETHOD_DEFAULT = 0,
    D3DDECLMETHOD_PARTIALU,
    D3DDECLMETHOD_PARTIALV,
    D3DDECLMETHOD_CROSSUV,
    D3DDECLMETHOD_UV,
    D3DDECLMETHOD_LOOKUP,
    D3DDECLMETHOD_LOOKUPPRESAMPLED
} D3DDECLMETHOD;
```

The Usage and UsageIndex members describe the semantics of how the element is used during rendering. The semantic for a vertex element is used to map vertex elements to vertex shader registers and inputs to the fixed-function pipeline. The Usage member is a value from the D3DDECLUSAGE enumeration. The UsageIndex is used to distinguish multiple elements of the same use semantic, such as a texture coordinate set for texture stage 0 and a second texture coordinate set for teture stage 1.

```
typedef enum _D3DDECLUSAGE
{
    D3DDECLUSAGE_POSITION = 0,
    D3DDECLUSAGE_BLENDWEIGHT,
    D3DDECLUSAGE_BLENDINDICES,
    D3DDECLUSAGE_NORMAL,
    D3DDECLUSAGE_PSIZE,
    D3DDECLUSAGE_TEXCOORD,
    D3DDECLUSAGE_TANGENT,
    D3DDECLUSAGE_BINORMAL,
    D3DDECLUSAGE_TESSFACTOR,
    D3DDECLUSAGE_POSITIONT,
    D3DDECLUSAGE_COLOR,
    D3DDECLUSAGE_FOG,
    D3DDECLUSAGE_DEPTH,
    D3DDECLUSAGE_SAMPLE,
} D3DDECLUSAGE;
```

Most of these usage semantics correspond to elements of the fixed-function pipeline, but some are new to the shader models supported in DirectX 9.0c. D3DDECLUSAGE_POSITION is for a position in model coordinates requiring vertex processing, while D3DDECLUSAGE_POSITIONT is for a position that is already been transformed into screen space. The D3DDECLUSAGE_BLENDWEIGHT, D3DDECLUSAGE_BLENDINDICES, D3DDECLUSAGE_NORMAL, D3DDECLUSAGE_PSIZE, D3D-

DECLUSAGE_TEXCOORD, and D3DDECLUSAGE_COLOR semantics have familiar names from the corresponding fields in an FVF code.

The remaining usage semantics indicate additional usages that are convenient for vertex and pixel shaders. If you have custom per-vertex data that is not represented by the existing usage semantics, you can always treat it as an additional texture coordinate on each vertex. However, it is best to instruct the runtime on the exact nature of your data if there is an existing semantic that matches your data.

D3DDECLUSAGE_TANGENT indicates that the data contains a vector that is tangent to the underlying surface, instead of vector that is normal to the underlying surface. D3DDECLUSAGE_BINORMAL indicates that the data contains a vector that is the binormal to the underlying surface.

A per-vertex tessellation factor consisting of a single `float` is designated with D3DDECLUSAGE_TESSFACTOR. D3DDECLUSAGE_SAMPLE describes per-vertex tessellator sampler data and is always used with D3DDECLMETHOD_LOOKUP and D3DDECLMETHOD_LOOKUPPRESAMPLED.

The D3DDECLUSAGE_FOG and D3DDECLUSAGE_DEPTH semantics have been created to supply fog-factor and fog depth values as outputs from vertex shaders so that pixel shaders can compute the appropriate fog effects if necessary.

Here are some sample vertex structures and their corresponding vertex element arrays.

```
// position, diffuse color, normal
struct s_vertex1
{
D3DVECTOR position;
D3DVECTOR normal;
D3DCOLOR diffuse;
};

D3DVERTEXELEMENT9 declaration1[] =
{
{ 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 0 },
{ 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL, 0 },
{ 0, 24, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_COLOR, 0 },
D3DDECL_END()
};

// tweened vertices:
// stream 0: position1, normal1, diffuse
// stream 1: postiion2, normal2
struct s_stream0
{
```

```
D3DVECTOR position;
D3DVECTOR normal;
D3DCOLOR diffuse;
};
struct s_stream1
{
D3DVECTOR position;
D3DVECTOR normal;
};

D3DVERTEXELEMENT9 declaration2[] =
{
{ 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 0 },
{ 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL, 0 },
{ 0, 24, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_COLOR, 0 },
{ 1, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 1 },
{ 1, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL, 1 },
D3DDECL_END()
};
```

Vertex declarations are represented in the runtime by the `IDirect3DVertexDeclaration9` interface, summarized in interface 5.1. The interface only has two read-only properties: the associated device and the associated vertex element array. When retrieving the vertex element array with `GetDeclaration`, you can determine the size of the array needed by passing in `NULL` for the `result` parameter and the address of the variable to be filled with the `count` parameter. The immutability of a vertex declaration makes it easy for the runtime to manage the use of the declarations. When you need to change a vertex declaration, simply release the existing declaration and create a new one. Creating vertex declarations is a fast operation in order to support this model.

Interface 5.1: Summary of the `IDirect3DVertexDeclaration9` interface.

IDirect3DVertexDeclaration9

Read-Only Properties	
`GetDeclaration`	The shader declaration token array.
`GetDevice`	The associated device.

```
interface IDirect3DVertexDeclaration9 : IUnknown
```

```
{
 //-----------------------------------------------------------
 // read-only properties
 HRESULT GetDevice(IDirect3DDevice9 **result);
 HRESULT GetDeclaration(D3DVERTEXELEMENT9 *result,
UINT *count);
};
```

You obtain an instance of this interface by passing an array of vertex element structures to the `CreateVertexDeclaration` method. Vertex declarations always reside in system memory and don't need to be restored when a device is lost.

```
HRESULT CreateVertexDeclaration(CONST D3DVERTEXELEMENT9 *elements,
IDirect3DVertexDeclaration9 **result);
```

The vertex declaration property on the device is used to declare vertices with a declaration instead of an FVF code. The `GetVertexDeclaration` and `SetVertexDeclaration` methods are used to manipulate the property.

```
HRESULT GetVertexDeclaration(IDirect3DVertexDeclaration9 **value);
HRESULT SetVertexDeclaration(IDirect3DVertexDeclaration9 *value);
```

5.8.1 Fixed-Function Declarations

When using the fixed-function pipeline with a vertex declaration, the appropriate vertex components must each be mapped to a particular usage. You can use multiple streams, but the vertices must conform to the order and type constraints of the FVF codes described in section 5.7. For instance, the following declares a vertex with two streams.

```
struct stream0
{
    D3DCOLOR m_diffuse;
    D3DVECTOR m_pos;
    D3DVECTOR m_normal;
};

struct stream1
{
    float m_u;
    float m_v;
};

const D3DVERTEXELEMENT9 decl[] =
{
{ 1, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT,
```

Vertex Component	Usage	Usage Index
Position	D3DDECLUSAGE_POSITION	0
Transformed Position	D3DDECLUSAGE_POSITIONT	0
Blend Weights	D3DDECLUSAGE_BLENDWEIGHT	0
Blend Weight Indices	D3DDECLUSAGE_BLENDINDICES	0
Normal	D3DDECLUSAGE_NORMAL	0
Point Size	D3DDECLUSAGE_PSIZE	0
Diffuse Color	D3DDECLUSAGE_COLOR	0
Specular Color	D3DDECLUSAGE_COLOR	1
Texture Coordinate 0	D3DDECLUSAGE_TEXCOORD	0
Texture Coordinate 1	D3DDECLUSAGE_TEXCOORD	1
Texture Coordinate 2	D3DDECLUSAGE_TEXCOORD	2
Texture Coordinate 3	D3DDECLUSAGE_TEXCOORD	3
Texture Coordinate 4	D3DDECLUSAGE_TEXCOORD	4
Texture Coordinate 5	D3DDECLUSAGE_TEXCOORD	5
Texture Coordinate 6	D3DDECLUSAGE_TEXCOORD	6
Texture Coordinate 7	D3DDECLUSAGE_TEXCOORD	7
Texture Coordinate 8	D3DDECLUSAGE_TEXCOORD	8
Tween Position 2	D3DDECLUSAGE_POSITION	1
Tween Normal 2	D3DDECLUSAGE_NORMAL	1

Table 5.2: Vertex usage for the fixed-function pipeline.

```
D3DDECLUSAGE_TEXCOORD, 0 },
{ 0, 0, D3DDECLTYPE_D3DCOLOR, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_COLOR, 0 },
{ 0, 4, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL, 0 },
{ 0, 16, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION, 0 },
D3DDECL_END()
};
```

The fixed-function pipeline requires a declaration that maps vertex components to a specific usage and index. The required usages are given in table 5.2. The values map to the appropriate usage and index as you would expect, with most of the values mapping to usage index zero. Texture coordinate sets are mapped to the usage index that corresponds to their texture stage. The diffuse and specular colors map to the usage indices zero and one, respectively. Vertex tweening uses a position and normal component at index zero and one.

5.9 Vertex Buffers

Vertex buffer resources are used to store the application's vertex data for defining primitives in a scene. Direct3D exposes vertex buffer resources through the

IDirect3DVertexBuffer9 interface, summarized in interface 5.2.

Interface 5.2: Summary of the IDirect3DVertexBuffer9 interface.

IDirect3DVertexBuffer9

Read-Only Properties	
GetDesc	A description of the contained vertex data.
Methods	
Lock	Obtains direct access to the contained vertex data.
Unlock	Releases direct access to the contained vertex data.

```
interface IDirect3DVertexBuffer9 : IUnknown
{
 //-------------------------------------------------------------
 // read-only properties
 HRESULT GetDesc(D3DVERTEXBUFFER_DESC *value);

 //-------------------------------------------------------------
 // methods
 HRESULT Lock(UINT offset,
              UINT size,
              BYTE **data,
              DWORD flags);
 HRESULT Unlock();
};
```

A vertex buffer containing size bytes of data can be created with CreateVertexBuffer.

```
HRESULT CreateVertexBuffer(UINT size,
             DWORD usage,
             DWORD fvf,
             D3DPOOL pool,
             IDirect3DVertexBuffer9 **result,
             HANDLE *unused);
```

If the fvf parameter is not zero, then the vertex buffer must have a valid FVF as described in the previous section and the size argument must be large enough to hold at least one vertex. If the fvf parameter is zero, then a non-FVF vertex buffer is created and the contents of the buffer must be described to Direct3D with a vertex shader declaration, as described in chapter 9. The pool argument gives the memory pool for the vertices. The unused argument must be NULL. The usage argument can be zero or more of the following flags.

```
#define D3DUSAGE_DONOTCLIP           0x00000020L
#define D3DUSAGE_DYNAMIC             0x00000200L
#define D3DUSAGE_NPATCHES            0x00000100L
#define D3DUSAGE_POINTS              0x00000040L
#define D3DUSAGE_RTPATCHES           0x00000080L
#define D3DUSAGE_SOFTWAREPROCESSING 0x00000010L
#define D3DUSAGE_WRITEONLY           0x00000008L
```

D3DUSAGE_DONOTCLIP indicates that the vertices will not need **clipping**. Clipping is described in more detail in chapter 7. D3DUSAGE_NPATCHES, D3DUSAGE_POINTS, and D3DUSAGE_RTPATCHES indicate that the vertex buffer will be used to draw N-patches, point sprites, or higher order surface patches, respectively. D3DUSAGE_SOFTWAREPROCESSING indicates that the vertex buffer will be used with software vertex processing.

D3DUSAGE_WRITEONLY tells Direct3D that the application will never read from the vertex buffer, only write to the buffer. This allows the runtime and the driver to be more efficient in providing vertex buffer space for the application. Attempts to read from a write-only vertex buffer will fail.

D3DUSAGE_DYNAMIC indicates that the application will dynamically change the contents of the vertex buffer. This is in contrast to a "static" vertex buffer where the application loads the contents of the buffer just once when the resource is created. If D3DUSAGE_DYNAMIC is not specified, then the vertex buffer is static. An application typically uses static vertex buffers for models that don't change shape and don't need their vertices redefined once they are loaded. If you are generating procedural geometry, performing software deformation on geometry, or any other sort of vertex "editing", then you will need a dynamic vertex buffer for that data.

If the D3DDEVCAPS_TLVERTEXVIDEOMEMORY bit or the D3DDEVCAPS_TLVERTEXSYSTEMMEMORY bit of D3DCAPS9::DevCaps are set, then the device can use vertex buffers containing transformed vertices from video or system memory, respectively.[1]

```
#define D3DDEVCAPS_TLVERTEXSYSTEMMEMORY 0x00000040L
#define D3DDEVCAPS_TLVERTEXVIDEOMEMORY  0x00000080L
#define D3DDEVCAPS_EXECUTESYSTEMMEMORY  0x00000010L
#define D3DDEVCAPS_EXECUTEVIDEOMEMORY   0x00000020L
```

In general, static vertex buffers end up in device memory, while dynamic vertex buffers will be placed in memory readily accessed by the CPU, such as system memory or AGP memory. Dynamic vertex buffers must be allocated in either the system memory pool or in the default memory pool. Attempts to create a dynamic vertex buffer in the managed memory pool will fail.

GetDesc returns a description of the vertex buffer in a D3DVERTEXBUFFER_DESC structure with Format, Type, Usage, and Pool members as described in

[1]The D3DDEVCAPS_EXECUTEVIDEOMEMORY and D3DDEVCAPS_EXECUTESYSTEMMEMORY flags are similar and indicate the device supports execute buffers from video or system memory, respectively. Execute buffers are not exposed by the API, these bits are only informational.

section 2.7. Format will always be D3DFMT_VERTEXDATA for vertex buffer resources. The FVF member gives the fvf argument passed to CreateVertexBuffer.

```
typedef struct _D3DVERTEXBUFFER_DESC
{
    D3DFORMAT       Format;
    D3DRESOURCETYPE Type;
    DWORD           Usage;
    D3DPOOL         Pool;
    UINT            Size;
    DWORD           FVF;
} D3DVERTEXBUFFER_DESC;
```

The Lock method provides direct access to the contained vertex data, and Unlock relinquishes access to that data, similar to the LockRect and UnlockRect methods on IDirect3DSurface9. The access to the vertex buffer's data must be consistent with the value of the flags argument to Lock, which can be zero or a combination of the following flags.

```
#define D3DLOCK_DISCARD     0x00002000L
#define D3DLOCK_NOOVERWRITE 0x00001000L
#define D3DLOCK_NOSYSLOCK   0x00000800L
#define D3DLOCK_READONLY    0x00000010L
```

D3DLOCK_DISCARD indicates that the application does not depend on the previous contents of the vertex buffer and that the entire previous contents can be discarded. D3DLOCK_NOOVERWRITE indicates that the application will not overwrite any vertices that have been used in drawing primitives since the start of the scene or the last lock on this vertex buffer. D3DLOCK_DISCARD and D3DLOCK_NOOVERWRITE can only be used with dynamic vertex buffers. D3DLOCK_NOSYSLOCK and D3DLOCK_READONLY are as described in section 4.3. For scenes with dynamic geometry, improper locking of dynamic vertex buffers can significantly impact rendering performance, see chapter 23.

The following excerpt from the rt_GingerBread.cpp file in the sample code locks a vertex buffer and fills it with dynamically generated point data.

```
{
    vertex_lock<s_vertex> lock(m_vb, D3DLOCK_DISCARD);
    s_vertex *points = lock.data();
    for (UINT i = 0; i < m_num_points; i++)
    {
        // store current
        points[i].set(x, y, m_fg);

        // compute next
        const float next_x = 1.0f - y + (x < 0.0f ? -x : x);
```

```
        y = x;
        x = next_x;

        // adjust bounding box
        accum_minmax(x, min_x, max_x);
        accum_minmax(y, min_y, max_y);
    }

    // forces lock to be released, holding it as little as possible
}
```

Every succesful call to `Lock` must be followed by a call to `Unlock`. A vertex buffer can be locked multiple times, provided that the same number of calls to both `Lock` and `Unlock` are made. A vertex buffer cannot be used by the device while it is locked.

The vertex data pointer returned by `Lock` is of type `BYTE **`, requiring the use of `reinterpret_cast<>` to obtain a pointer to your vertex data structure. We can create a vertex buffer locking helper class as we did for surfaces, but writing out explicit data accessors for all possible vertex data structures would be tedious and error-prone. We can use a template class where the vertex data type is passed as a parameter to avoid this problem. The header file `<rt/vertexbuf.h>` in the sample code contains the helper class in listing 5.1.

Listing 5.1: `<rt/vertexbuf.h>`: A vertex buffer lock helper class.

```
1  #if !defined(RT_VERTEXBUF_H)
2  #define RT_VERTEXBUF_H
3  //------------------------------------------------------------
4  // vertexbuf.h
5  //
6  // Helper classes for use with vertex buffers.
7  //
8  #include <atlbase.h>
9
10 namespace rt {
11
12 //------------------------------------------------------------
13 // vertex_lock
14 //
15 // Locks a vertex buffer in its constructor and unlocks it
16 // in its destructor, providing for exception safe access
17 // to vertex data.  Takes a template argument Vertex that
18 // is the type of the underlying vertices to be locked.
19 //
20 template <typename Vertex>
21 class vertex_lock
22 {
```

```
23  public:
24      // lock in constructor, unlock in destructor
25      vertex_lock(IDirect3DVertexBuffer9 *vb,
26          DWORD flags = 0,
27          UINT offset = 0,
28          UINT size = 0)
29          : m_vb(vb)
30      {
31          void *data = 0;
32          THR(m_vb->Lock(offset, size, &data, flags));
33          m_data = static_cast<Vertex *>(data);
34      }
35      ~vertex_lock()
36      {
37          const HRESULT hr = m_vb->Unlock(); hr;
38          ATLASSERT(SUCCEEDED(hr));
39      }
40
41      // type safe accessors to vertex data
42      const Vertex *data() const { return m_data; }
43      Vertex *data() { return m_data; }
44
45  private:
46      CComPtr<IDirect3DVertexBuffer9> m_vb;
47      Vertex *m_data;
48  };
49
50  }; // rt
51
52  #endif
```

5.10 Indexed Primitives

In section 5.5 we saw that even a simple shape may require vertices to be duplicated in order to satisfy the desired winding order and primitive topology constraints. If we assign each vertex an index and send a list of unique vertices and a list of indices to be used for composing the primitive, we can reduce the amount of data that is sent to the device.

Referring back to figure 5.3, we have repeated vertices for `D3DPT_LINELIST` and `D3DPT_TRIANGLELIST`. Using an indexed primitive, we could draw figure 5.3(b) using `D3DPT_LINELIST` with the vertices as given in figure 5.3(d) and the indices {0, 2, 2, 12, 12, 13, 13, 17, 17, 14, 14, 10, 10, 11, 11, 4, 4, 0}. Notice that we didn't refer to every index in the table, just as when we listed the vertices explicitly. We could draw figure 5.3(c) using `D3DPT_TRIANGLELIST`

and the indices {0, 1, 3, 1, 4, 3, 1, 2, 4, 2, 5, 4, 4, 5, 11, 5, 12, 11, 10, 11, 14, 11, 15, 14, 11, 12, 15, 12, 16, 15, 12, 13, 16, 13, 17, 16}.

Indexed primitives gain over non-indexed primitives when the size of the vertex is large relative to the size of the indices and sharing of vertices is common. This is usually the case for typical models. All primitive types except `D3DPT_POINTLIST` can be drawn as indexed primitives.

5.10.1 Index Buffers

Index buffer resources are used to store indices into the application's vertex data for defining primitives in a scene. Direct3D exposes index buffer resources through the `IDirect3DIndexBuffer9` interface, summarized in interface 5.3.

Interface 5.3: Summary of the `IDirect3DIndexBuffer9` interface.

IDirect3DIndexBuffer9

Read-Only Properties	
`GetDesc`	A description of the contained index data.
Methods	
`Lock`	Obtains direct access to the contained index data.
`Unlock`	Releases direct access to the contained index data.

```
interface IDirect3DIndexBuffer9 : IUnknown
{
 //-------------------------------------------------------------
 // read-only properties
 HRESULT GetDesc(D3DINDEXBUFFER_DESC *value);

 //-------------------------------------------------------------
 // methods
 HRESULT Lock(UINT offset,
              UINT size,
              BYTE **data,
              DWORD flags);
 HRESULT Unlock();
};
```

An index buffer containing `size` bytes of data is created by calling the `CreateIndexBuffer` method on the device.

```
HRESULT CreateIndexBuffer(UINT size,
             DWORD usage,
             D3DFORMAT format,
```

```
            D3DPOOL pool,
            IDirect3DIndexBuffer9 **result,
            HANDLE *unused);
```

The `usage` and `pool` arguments are as described for vertex buffers in section 5.9. The `format` argument must be either `D3DFMT_INDEX16` or `D3DFMT_INDEX32` indicating `WORD` indices or `DWORD` indices, respectively. The `unused` argument must be `NULL`.

`GetDesc` returns a description of the index buffer in a `D3DINDEXBUFFER_DESC` structure. The members of this structure are as described in section 2.7.

```
typedef struct _D3DINDEXBUFFER_DESC
{
    D3DFORMAT       Format;
    D3DRESOURCETYPE Type;
    DWORD           Usage;
    D3DPOOL         Pool;
    UINT            Size;
} D3DINDEXBUFFER_DESC;
```

The `Lock` method provides direct access to the contained index data, and `Unlock` relinquishes access to that data in a manner identical to the `Lock` and `Unlock` methods on `IDirect3DVertexBuffer9`, described in section 5.9. The sample code contains a locking helper class, similar to that presented for vertex buffers, in the file `<rt/indexbuf.h>`.

5.11 The Vertex Shader

The vertex shader property of the device selects between fixed-function vertex processing and programmable vertex processing. The property is a vertex shader interface pointer manipulated with the `SetVertexShader` and `GetVertexShader` methods. Like all device properties exposed as interfaces, the device will add a reference on any vertex shader set on the device. Setting the property to `NULL` selects fixed-function processing and a valid interface pointer selects programmable vertex processing as described in chapter 9.

```
HRESULT GetVertexShader(IDirect3DVertexShader9 **value);
HRESULT SetVertexShader(IDirect3DVertexShader9 *value);
```

Programmable vertex shaders are created with `CreateVertexShader`. When the vertex shader is no longer needed, release the vertex shader COM object and ensure that it is not currently set on the device.

```
HRESULT CreateVertexShader(const DWORD *function,
            IDirect3DVertexShader9 **result);
```

The IDirect3DVertexShader9 interface is summarized in interface 5.4. The GetDevice method returns the device associated with the shader and the GetFunction method returns the shader definition. To obtain the definition, first call GetFunction with NULL for the output buffer to obtain the size, allocate enough memory to hold the data, then call the function again to obtain the data. GetFunction will fail if the size argument is NULL or the size is not large enough to hold the vertex shader function.

Interface 5.4: Summary of the IDirect3DVertexShader9 interface.

IDirect3DVertexShader9

Read-Only Properties	
GetDevice	The associated device.
GetFunction	The shader function tokens.

```
interface IDirect3DVertexShader9 : IUnknown
{
 //-------------------------------------------------------------
 // read-only properties
 HRESULT GetDevice(IDirect3DDevice9 **value);
 HRESULT GetFunction(void *value, DWORD *size);
};
```

Generally an application would not create the DWORD function array directly, but would use D3DX to compile an assembly language shader or a high level shader and pass the resulting array to the runtime. Assembly language vertex shaders are discussed in chapter 9; high level vertex shaders are discussed in chapter 18.

5.12 Drawing Primitives

Once all the scene data has been defined and vertex processing configured, the application can cause rendering to occur by calling one of the draw methods on the device: DrawPrimitiveUP, DrawIndexedPrimitiveUP, DrawPrimitive, DrawIndexedPrimitive, DrawTriPatch or DrawRectPatch.

```
HRESULT DrawPrimitiveUP(D3DPRIMITIVETYPE kind,
            UINT primitive_count,
            const void *vertex_data,
            UINT vertex_stride);
```

DrawPrimitiveUP draws non-indexed primitives of type kind from the supplied vertex data. The vertex data is arranged in memory such that successive

Primitive Type	Vertices	Primitives
D3DPT_POINTLIST	n	n
D3DPT_LINELIST	$2n$	n
D3DPT_LINESTRIP	$n+1$	n
D3DPT_TRIANGLEFAN	$n+2$	n
D3DPT_TRIANGLESTRIP	$n+2$	n
D3DPT_TRIANGLELIST	$3n$	n

Table 5.3: Relationship of primitive count to vertex count for the Direct3D primitive types.

vertices are spaced `vertex_stride` bytes apart. The `primitive_count` argument gives the number of primitives drawn by the call. The number of vertices corresponding to the number of primitives for different primitive types is given in table 5.3.

Internally, `DrawPrimitiveUP` copies the supplied data from the caller's memory into a new vertex buffer, issues the primitives using that vertex buffer, and then releases the buffer. The construction and destruction of a vertex buffer and extra copy of the primitive data on each call to `DrawPrimitiveUP` can be a costly operation, but this method can be convenient for experimenting or rendering low polygon count scenes. The `minimal.cpp` application presented in chapter 2, uses `SetVertexShader` and `DrawPrimitiveUP` on lines 56–58.

```
HRESULT DrawIndexedPrimitiveUP(D3DPRIMITIVETYPE kind,
            UINT min_index,
            UINT num_vertices,
            UINT primitive_count,
            const void *index_data,
            D3DFORMAT index_format,
            const void *vertex_data,
            UINT vertex_stride);
```

`DrawIndexedPrimitiveUP` operates similarly to `DrawPrimitiveUP`, but draws indexed primitives using the supplied vertex and index data. The `primitive_count`, `kind`, `vertex_data`, and `vertex_stride` arguments are the same as in `DrawPrimitiveUP`. The `min_index` argument gives the minimum vertex index used for the primitives drawn by the call, where index zero is at the location of `vertex_data`. The `num_vertices` argument gives the number of vertices used for the primitives drawn by the call, relative to `min_index`. For instance, if an application uses vertices at indices 3-9, then `min_index` is 3 and `num_vertices` is 6. The `index_data` argument must point to an array of indices used in the primitives and `index_format` arguments give the the format of the indices (`WORD` or `DWORD`).

5.13 Vertex Data Streams

Vertex components are gathered from one or more **streams** and concatenated together to assemble a complete vertex. The assembled vertex is the starting point for all Direct3D pipeline processing. Each stream is numbered with an unsigned integer and associated with a vertex buffer source and a stride. The streams are numbered consecutively, starting with zero, and are concatenated in order, beginning with stream zero, to form an entire vertex. Streams must contain an integral number of vertex components; a vertex component can't be split across a stream boundary.

Assigning different streams to different vertex components allows an application to dynamically change the values of vertex components in a model without locking the entire vertex buffer containing all the components only to change a portion of the vertex buffer. With multiple streams, the static components of the vertices can be retrieved from one set of streams and the dynamic components retrieved from another set of streams. This allows the application to use the `D3DLOCK_DISCARD` flag on the vertex buffers containing the dynamic components.

A stream is associated with a vertex buffer by setting the stream source property with `SetStreamSource`. The `stride` argument is the size of the vertex components stored in the vertex buffer associated with the stream. For an FVF vertex buffer, this must be the same as the size of the vertex. For a non-FVF vertex buffer, the size must be at least as big as the size of the vertex computed from its vertex declaration, see chapter 9 for more about vertex shader declarations. The value of the stream source property can be retrieved with `GetStreamSource`.

```
HRESULT GetStreamSource(UINT index,
            IDirect3DVertexBuffer9 **value,
            UINT *offset,
            UINT *stride);
HRESULT SetStreamSource(UINT index,
            IDirect3DVertexBuffer9 *value,
            UINT offset,
            UINT stride);
```

The `offset` argument provides an byte offset from the beginning of the vertex buffer for the first vertex in the stream. This allows a single vertex buffer to be split between multiple vertex types simultaneously by treating them as two chunks of memory within the same buffer, one offset from the other. If a device supports stream offsets, then the `D3DDEVCAPS2_STREAMOFFSET` bit will be set in `D3DCAPS9::Caps2`.

```
#define D3DDEVCAPS2_STREAMOFFSET 0x00000001L
```

The maximum number of simultaneous streams that can be used with the device is given by `D3DCAPS9::MaxStreams`. For a DirectX 8.1 or later driver,

this will be a value between 1 and 16. For a pre-DirectX 8.1 driver, the value will be zero indicating that the device can only use a single stream. The `MaxStreamStride` member gives the maximum stride that can be set for any stream.

Stream sources provide vertex data, but we must also supply index data from an index buffer for indexed primitives. `SetIndices` sets the given index buffer for use with indexed primitives. While there can be up to 16 vertex data streams, there is only a single index buffer used by all the data streams. The `GetIndices` method returns the value of the indices property.

```
HRESULT GetIndices(IDirect3DIndexBuffer9 **value);
HRESULT SetIndices(IDirect3DIndexBuffer9 *value);
```

With all the vertex components and indices loaded into resources, fixed-function vertex processing selected, and the stream source set appropriately, all the necessary state has been set for a call to `DrawPrimitive` or `DrawIndexedPrimitive` to draw non-indexed or indexed primitives, respectively.

```
HRESULT DrawPrimitive(D3DPRIMITIVETYPE kind,
            UINT start_vertex,
            UINT primitive_count);
```

`DrawPrimitive` draws non-indexed primitives of the given `kind`. The `start_vertex` argument gives the index of the first vertex, taken from the currently set streams, defining the primitives. The streams must hold enough vertex data to define the requested number of primitives.

```
HRESULT DrawIndexedPrimitive(D3DPRIMITIVETYPE kind,
            UINT base_vertex_index,
            UINT min_index,
            UINT num_vertices,
            UINT start_index,
            UINT primitive_count);
```

`DrawIndexedPrimitive` draws indexed primitives of the given `kind`. The `min_index` argument gives the minimum vertex index used by any of the drawn primitives and is relative to the base vertex index of the indices. The `start_index` argument gives an offset into the indices, relative to the base vertex index of the indices, from which to read indices for the primitives. The `base_vertex_index` argument is added to all indices retrieved from the index buffer before indexing the vertices. This allows multiple primitives to be packed into a set of vertex data streams without requiring that the indices be changed based on the location of the vertex data in the vertex buffer. See figure 5.4 for a graphical illustration of the relationship between the `start_index`, `primitive_count`, `base_vertex_index`, `min_index`, and `num_vertices` arguments to `DrawIndexedPrimitive`. The remaining arguments are as in `DrawPrimitive`.

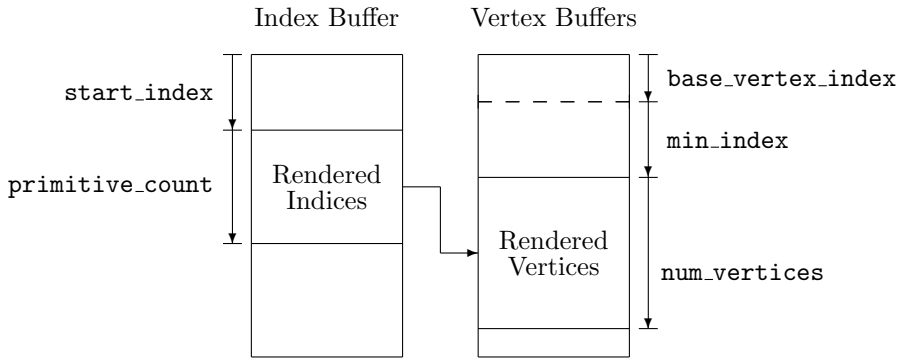


Figure 5.4: Relationship between arguments to `DrawIndexedPrimitive`. The appropriate number of indices for `primitive_count` primitives are used to index `num_vertices` vertices in the vertex buffer relative to `base_vertex_index` and `min_index`.

As an example, suppose we have 21 distinct vertices consisting of the origin and 20 points arranged at equal angles around a circle.

$$\begin{aligned} P_i &= \left(\cos\left(\tfrac{\pi i}{10}\right), \sin\left(\tfrac{\pi i}{10}\right), 0\right),\ i = 0, \ldots, 19 \\ P_{20} &= (0, 0, 0) \end{aligned}$$

The index buffer is filled with 60 indices to define 20 triangles in the topology of a triangle list.

$$I_{3k} = 20,\ I_{3k+1} = k + 1,\ I_{3k+2} = k,\ k = 0, \ldots, 19$$

Now we can draw a pie shape from the 20-gon defined by the vertices. If we want to draw the last triangle wedge using an indexed triangle list, we would draw the triangle $P_{20}P_{19}P_{18}$. The `base_vertex_index` locates the beginning of the block of 21 vertices in the vertex buffer, and the call to `DrawIndexedPrimitive` would be:

```
device->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 18, 3, 57, 1);
```

5.14 Capabilities for Vertex Assembly

The draw primitive methods of the device allow for the batching of primitives into a single method call. This is the most efficient way to send primitives to the device. Batching primitives to the device allows for more concurrency between the device and the CPU: while the device is rasterizing one batch of primitives, the CPU is preparing and issuing the next batch. Batching strategies are discussed in chapter 23.

The `MaxPrimitiveCount` member of `D3DCAPS9` gives the maximum number of primitives that can be issued in any single draw primitive method. The `MaxVertexIndex` member gives the largest vertex index that may be used with the device. If this number is larger than $2^{16} - 1$, then 32 bit indices are supported on the device. Referring to figure 5.4, because `base_vertex_index` is added to indices used for primitives, you should ensure that the sum is less than `MaxVertexIndex` for proper operation.

If the `D3DDEVCAPS_DRAWPRIMTLVERTEX` bit of the `DevCaps` member is set, then the device has hardware support for `DrawPrimitive`. The `DrawPrimitive` method itself is always supported, this bit only indicates that the method is directly supported by the device's hardware abstraction layer interface. This flag is informational only, an application is not expected to change its behavior based on the value of this flag.

```
#define D3DDEVCAPS_DRAWPRIMTLVERTEX 0x00000400L
```

5.15 Enhanced Primitives

TODO: update this description

Direct3D also provides two enhancements to the basic primitive types described earlier in this chapter. Point sprites extend point primitives to have a screen space size that is rasterized with a texture. N-Patches provide a way to enhance the visual appearance of existing triangle models without significant rework to either the models or the program.

5.15.1 Point Sprites

Point sprites extend `D3DPT_POINTLIST` primitives to cover more than a single pixel when they are rasterized. If the `D3DCAPS9::MaxPointSize` member is greater than 1, then the device supports point sprites.[2] Point sprites are rendered as a textured square in screen space. The point size can be specified in the vertex buffer for each vertex with `D3DFVF_POINT_SIZE`. The structure of a point sprite is shown in figure 5.5.

If the point sprites are specified with `D3DFVF_XYZRHW`, then the size of the point sprite is given in screen space coordinates, otherwise the point sprite size is subject to the interpretation given in chapter 10.

Point sprites can be used to render many instances of a symbol by storing an image of the symbol in the texture and issuing point sprite primitives at the location of the symbols. By expanding the point sprite into a textured square in the device, we can achieve the desired rendering at one fourth the vertex transfer cost into the device. Point sprites can also be used for rendering a particle system, where each rendered particle is a point sprite.

[2] A point size of 1 pixel is always supported via `D3DPT_POINTLIST`.

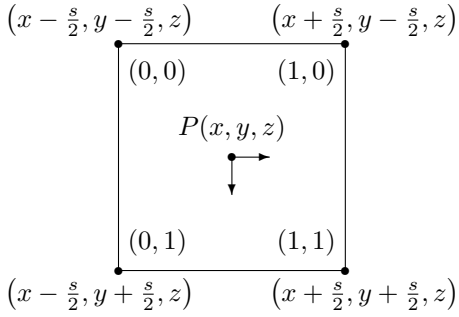


Figure 5.5: Screen space structure for a point sprite with vertex position P and size s. The texture coordinates at each computed sprite vertex are shown inside the square and the computed coordinates are shown outside the square.

5.15.2 *N*-Patches

N-Patches provide tessellation of triangles based on their position and surface normal information. If the `D3DDEVCAPS_NPATCHES` bit of `D3DCAPS9::DevCaps` is set, then the device supports N-patch tessellation of triangle primitives. In the absence of device support, an application can still use N-patch tessellation through `D3DXTessellateNPatches`, see chapter 19.

```
#define D3DDEVCAPS_NPATCHES 0x01000000L
```

For triangles approximating smooth surfaces, N-patches provide additional detail to the surface model without significant programming effort or changing model data. N-Patch tessellation is applied to triangles when the N-Patch mode property of the device is greater than 1.0. Additional triangles are generated by reconstructing a smooth surface patch from the source triangle position and normal information and then subdividing that patch according to the N-Patch mode property.

```
float GetNPatchMode();
HRESULT SetNPatchMode(float value);
```

Separate patches are generated for the vertex position and for the vertex normal. The algebraic degree of the patch for the positions of new vertices is defined by RS Position Degree. The degree of the patch for the normals of the new triangle is defined by RS Normal Degree. Both render states are values of the `D3DDEGREETYPE` enumeration. RS Position Degree can be set to linear or cubic, with the latter being the default. RS Normal Degree can be set to linear or quadratic, with the former being the default.

```
typedef enum _D3DDEGREETYPE
{
   D3DDEGREE_LINEAR    = 1,
```

```
   D3DDEGREE_QUADRATIC = 2,
   D3DDEGREE_CUBIC     = 3,
   D3DDEGREE_QUINTIC   = 5
} D3DDEGREETYPE;
```

5.16 Higher Order Surfaces

Triangular and rectangular patches provide the highest quality parametrically defined primitive and allow an application to specify truly smooth surfaces to Direct3D, as opposed to triangulated approximations to the true surface. The surfaces are called "higher order" because the algebraic order of the equations describing the surface is higher than the algebraic order of the corresponding equations for a triangle.

Higher order surfaces, also called spline surfaces, have a rich body of research and algorithms covering their uses in computer graphics. The full mathematics of higher order surface patches is too complex to go into here; the interested reader is encouraged to examine the literature on splines for more detailed information. Section 5.18 provides a starting point.

With a triangle, we provide the vertices that define the triangle and that's all there is to it. With a patch, we provide a collection of vertices, called control points, and a patch definition that says how the smooth surface is generated from the control points. The surface is generated by taking a weighted sum of algebraic functions that are influenced by the control points. The collection of algebraic functions used to define each surface patch are called the **basis functions** for the patch.

Every patch is characterized by the basis functions used to construct the patch surface and the algebraic order of the basis functions. The `D3DBASISTYPE` and `D3DORDERTYPE` enumerations specify the basis and order of patches, respectively. The Bézier basis provides for a smooth surface that comes near the control points but only passes through the control points at the corners of the patch definition. The B-spline basis provides for a smooth surface that does not necessarily pass through any of the control points. The interpolating basis provides for a surface that is guaranteed to pass through all of the control points.

```
typedef enum _D3DBASISTYPE
{
   D3DBASIS_BEZIER      = 0,
   D3DBASIS_BSPLINE     = 1,
   D3DBASIS_INTERPOLATE = 2
} D3DBASISTYPE;
```

Direct3D handles higher order surfaces by tessellating the surface into a collection of triangles that can be further processed by the remainder of the pipeline. The tessellated triangulation can be cached by associating the tessel-

lation with a patch handle and reusing the handle on subsequent renderings of the patch with the same tessellation.

If the D3DDEVCAPS_RTPATCHES bit of D3DCAPS9::DevCaps is set, the device supports RT patches with the DrawTriPatch and DrawRectPatch methods. If D3DDEVCAPS_QUINTICRTPATCHES bit is set, the device supports quintic Bézier and B-spline RT patches. If the D3DDEVCAPS_PATCHHANDLEZERO bit is set, it indicates that uncached patch handles will be drawn as efficiently as cached patch handles.

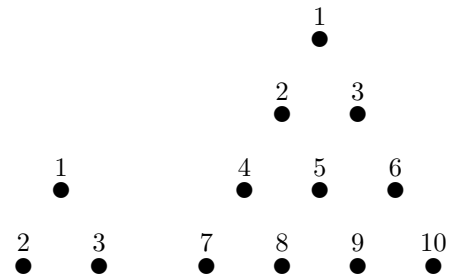
```
#define D3DDEVCAPS_QUINTICRTPATCHES  0x00200000L
#define D3DDEVCAPS_RTPATCHES         0x00400000L
#define D3DDEVCAPS_RTPATCHHANDLEZERO 0x00800000L
```

Triangular and rectangular patches are both tessellated and rendered with the DrawTriPatch and DrawRectPatch methods, respectively. Tessellation of a patch description involves a fair amount of computation and it is useful to perform this only when the patch definition changes, and reuse the results on subsequent renderings. If the handle argument is not NULL, then the patch is tessellated according to the patch_info argument and the tessellation is associated with the supplied handle identifier, assigned by the caller. The tessellation can be drawn again by passing the same handle and number of segments and a patch_info argument of NULL. To change only the number of segments but reuse the same definition use NULL for the patch information and pass a pointer to the new data for the number of segments. If the patch definition changes, the patch can be re-tessellated into the same handle. When the cached tessellation is no longer needed, call DeletePatch on the handle to release the associated memory.

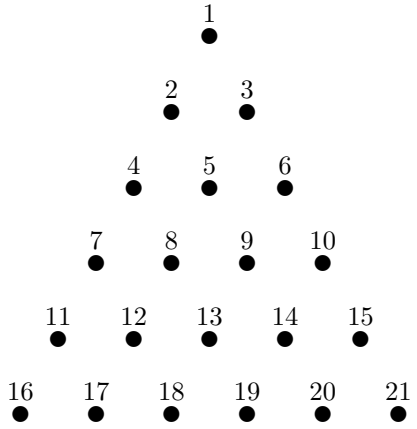
```
HRESULT DeletePatch(UINT handle);
HRESULT DrawRectPatch(UINT handle,
            const float *num_segments,
            const D3DRECTPATCH_INFO *patch_info);
HRESULT DrawTriPatch(UINT handle,
            const float *num_segments,
            const D3DTRIPATCH_INFO *patch_info);
```

The num_segments argument points to an array of 3 floats for a triangular patch and an array of 4 floats for a rectangular patch. This allows an application to tessellate some edges of a patch more finely than other edges. For instance, the foreground edges of a patch may be tessellated more to provide more foreground detail on the model.

The resulting tessellation may be rendered directly by the hardware, or may be triangulated and rendered as traditional triangles. The currently set vertex streams are ignored when drawing patches. When dynamically changing the number of segments for a patch, abrupt changes in surface appearance can be caused when only the integer part of the patch segment count is considered in creating the tessellation. This creates a so-called “popping” artifact that



(a) Linear (b) Cubic



(c) Quintic

Figure 5.6: Bézier triangular patch vertex order.

appears when the integer part of the segment count changes. RS Patch Edge Style determines whether or not the segment count is treated as a discrete integral value or a continuous value, which avoids the popping artifact.

```
typedef enum _D3DPATCHEDGESTYLE
{
   D3DPATCHEDGE_DISCRETE    = 0,
   D3DPATCHEDGE_CONTINUOUS  = 1
} D3DPATCHEDGESTYLE;
```

5.16.1 Triangular Patches

For `DrawTriPatch`, the `patch_info` argument is a pointer to a `D3DTRIPATCH_INFO` structure that defines the collection of triangular patches drawn. Direct-3D supports triangular patches with a Bézier basis of linear, cubic or quintic

Basis	Degree	Count
Bézier	Linear	3
	Cubic	10
	Quintic	21

Table 5.4: Vertex count for triangular Bézier patches.

degree. The vertices giving the control points for the patch are arranged within the vertex buffer as shown in figure 5.6.

`StartVertexOffset` gives the index of the first control point vertex in the current set of streams. You can pack multiple patches into a single set of streams by calling `DrawTriPatch` with a different `StartVertexOffset` for each patch. The `NumVertices` member gives the number of control point vertices for the patch and is one of the values in table 5.4. The only supported value for `Basis` is `D3DBASIS_BEZIER`. The `Degree` member can be `D3DDEGREE_LINEAR`, `D3DDEGREE_CUBIC` or `D3DDEGREE_QUINTIC`.

```
typedef struct _D3DTRIPATCH_INFO
{
    UINT          StartVertexOffset;
    UINT          NumVertices;
    D3DBASISTYPE  Basis;
    D3DDEGREETYPE Degree;
} D3DTRIPATCH_INFO;
```

5.16.2 Rectangular Patches

For `DrawRectPatch`, the `patch_info` argument is a pointer to a `D3DRECTPATCH_INFO` structure that defines the rectangular patch to be drawn. Direct3D supports rectangular patches with the basis and degree combinations shown in table 5.5. The `Basis` member must be one of the values `D3DBASIS_BEZIER`, `D3DBASIS_BSPLINE` or `D3DBASIS_CATMULL_ROM` for a rectangular patch. The `Degree` member must be `D3DDEGREE_LINEAR`, `D3DDEGREE_CUBIC` or `D3DDEGREE_QUINTIC`, depending on the basis. The `Width` and `Height` members are measured in vertices and give the size of a patch. The allowed values of the width and height for different basis and degree combinations are shown in table 5.5.

The `StartVertexOffsetWidth`, `StartVertexOffsetHeight` and `Stride` members describe the layout of the control point vertices for the patch in the currently bound streams. The control vertices are read in rows from top to bottom across the patch and from left to right across each row. The control points for the patch are described as a rectangle embedded within a larger rectangle of vertices. The width of this rectangle is given by the `Stride` member and is at least as large as the width of the patch. When the stride is larger than the width of the patch, the `StartVertexOffsetWidth` member gives the offset in vertices of the first control point within the row of the rectangle. The `StartVertexOffsetHeight` is the offset in vertices for the first row of the rectangle.

Basis	Degree	Width	Height
Bézier	Linear	2	2
	Cubic	4	4
	Quintic	6	6
B-Spline	Linear	1	1
	Cubic	3	3
	Quintic	5	5
Catmull-Rom	Cubic	3	3

Table 5.5: Rectangular patch vertex grid sizes.

```
typedef struct _D3DRECTPATCH_INFO
{
    UINT          StartVertexOffsetWidth;
    UINT          StartVertexOffsetHeight;
    UINT          Width;
    UINT          Height;
    UINT          Stride;
    D3DBASISTYPE  Basis;
    D3DDEGREETYPE Order;
} D3DRECTPATCH_INFO;
```

As with triangular patches, the `StartVertexOffsetWidth`, `StartVertexOffsetHeight` and `Stride` members of the rectangular patch description allow multiple patches to be packed into a single set of streams. If the patches start at index 0 and consist of 16 control points each, then the `StartVertexOffsetHeight` member will be $0, 16, 32, 48, \ldots$ for successive patches.

5.17 Object Approximations

The geometric primitives supported by Direct3D can model many things directly, but not everything is best modeled with vertices. As we will see when we examine texturing and the frame buffer, there are ways to shape an object using appearance information such as alpha channels so that we are not forced to model every small detail in an object using vertices.

One type of object used by almost every application, but conspicuously missing from Direct3D's primitive types, is text. Applications can draw vector text with line strips or line lists and a suitable vector font definition, such as the Hershey fonts. Other forms of text may be desired, such as screen-space text using a GDI font, or even extruding the text in one dimension to obtain text as a triangulated surface. The `rt_Text` sample demonstrates three ways of drawing text with Direct3D.

5.18 Further Reading

Curved PN Triangles, Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell.
Gives the exact mathematics of N-patch tessellation and how new triangles are computed.

An Introduction to Splines for Use in Computer Graphics & Geometric Modeling, Richard H. Bartels, John C. Beatty, Brian A. Barsky.
Covers spline modeling in computer graphics in detail.

Physically-Based Modeling for Computer Graphics, Ronen Barzel
Physically-based modeling uses a simulation of physics to govern the movement and deformation of modeled objects. This is an advanced level book that describes the mathematics and implementation of such a simulation system.

Texturing and Modeling: A Procedural Approach, David S. Ebert (Editor)
Procedural methods generate models from an algorithm, instead of being created by an artist in a modeling package. This book describes a variety of techniques for using algorithms to generate realistic and complex models and textures.

Real-Time Rendering, Tomas Möller and Eric Haines.
Describes many real-time simulation techniques that can be used with Direct3D.