Chapter 6

Vertex Transformations

> "There are two worlds; the world that we can measure with line and rule, and the world that we feel with our hearts and imaginations."
>
> Leigh Hunt: *Men, Women, and Books*, 1847

6.1 Overview

In the previous chapter, we saw how we could define models with vertex buffers, index buffers and primitives. We also mentioned that dynamic vertex components could be implemented with multiple streams.

If we wanted to move a model, we could edit its definition by locking the vertex buffer and modifying the appropriate vertex components. Direct3D provides the vertex processing pipeline to transform the model as an alternative to editing every vertex in the model. The applied transformations include translation, rotation and scaling of vertices.

"**Vertex processing**" refers to all the computations that occur for each vertex of each primitive rendered on the device. Each stage of the vertex processing section of the pipeline affects one or more components of a vertex as it is processed.

Direct3D allows for the application to choose between hardware vertex processing, software vertex processing or a combination of the two. It is also possible to instruct Direct3D to perform software vertex processing and capture the results of the processing without rendering any primitives.

A vertex transformation is represented as a 4x4 matrix of floats. Matrices performing the basic operations of translation, rotation, and scaling are presented.

When multiple transformations are applied, the order in which they are applied can change the outcome of the composite transformation. Examples of common composite transformations are given.

Vertex processing begins with the world transformation. The world transformation is applied to model space vertices to map them into world space. Transform properties of the device are introduced with the `GetTransform`, `SetTransform`, and `MultiplyTransform` methods.

Transformations can be used to implement a scene hierarchy. A scene hierarchy is useful for drawing rigid jointed figures and relative positioning of models. We describe a jointed robot arm as an example of a transformation hierarchy.

Multiple world transforms can be applied to a single vertex and the resulting transformed vertices combined in a weighted average in a technique called **vertex blending**. Direct3D provides a variety of vertex blending options which are summarized.

The simplest form of vertex blending is called **tweening**, where a single floating point value weights a vertex transformed by two world transformation matrices. The vertex components for vertex blending weights and their associated FVF code are given.

Generalized vertex blending supplies two or three blending weights per vertex allowing three or four matrices to be used for blending.

Indexed vertex blending allows for up to 4 different matrices to be used for each vertex. For non-indexed vertex blending the matrices for all vertices of a primitive are the same, only their blending weights are varied per-vertex. For indexed vertex blending, the indices vary per-vertex, allowing different matrices to be applied at different vertices.

In the remaining sections we describe the fog, face culling, user clip plane application, view frustum clipping, homogeneous divide, and viewport application stages of the pipeline. The viewing and projection stages are discussed in chapter 7 and the lighting stage is discussed in chapter 8.

6.2 Vertex Processing

As described in section 2.13, a device can be created with software, mixed or hardware vertex processing. When a device is created with mixed vertex processing, `GetSoftwareVertexProcessing` and `SetSoftwareVertexProcessing` control if the hardware or software is used to process vertices. When this render state is set to `TRUE`, software vertex processing is selected, otherwise hardware vertex processing is selected. This render state is always set to `FALSE` or `TRUE` when the device is created with hardware or software vertex processing, respectively. When this render state is changed, the current streams, current indices, and current vertex shader are reset to their default values and will need to be restored.

Vertex processing can be summarized as the following pipeline stages: world transformation, texture coordinate generation, texture transformation, view transformation, vertex fog, lighting, projection transformation, primitive assembly, face culling, user clip plane application, view frustum clipping, homogeneous divide and finally viewport mapping. The end result of all vertex processing is a so-called “transformed and lit” vertex, with a position component in screen

space, diffuse and specular colors, and up to eight sets of texture coordinates. This information is fed to the rasterization section of the pipeline where it is interpolated into a stream of pixels for each primitive. Vertices with D3DFVF_-XYZRHW position components skip all vertex processing and are fed directly to the rasterization section. The minimal program in listing 2.1 used transformed vertices to draw a triangle in screen space.

```
HRESULT ProcessVertices(UINT source_start_index,
            UINT destination_index,
            UINT vertex_count,
            IDirect3DVertexBuffer9 *destination,
            IDirect3DVertexDeclaration9 *declaration,
            DWORD flags);
```

The ProcessVertices method can be used to apply software vertex processing to the vertices set on the current stream using all the currently set pipeline state. The processed vertices are written into the vertex buffer given in the destination argument, which must be an FVF vertex buffer. The source_-start_index and vertex_count arguments identify the source range from the current streams that will be processed. The destination_index argument specifies the starting location in the destination vertex buffer to receive the processed vertices. The declaration argument gives the vertex declaration of the destination vertex buffer and must use a single stream. When the current vertex shader is shader model 3.0, then the vertex declaration must be supplied. Otherwise, the declaration argument can be NULL and the FVF code associated with the output vertex buffer will be used for the output vertices. The flags argument may be zero or D3DPV_DONOTCOPYDATA to avoid copying vertex data not affected by vertex processing. The flags value can be bitwise orred with one of the D3DLOCK flags to specify additional locking semantics for the output buffer.

ProcessVertices will fail if either the input vertex buffer streams or the destination vertex buffer was not created with D3DUSAGE_SOFTWAREPROCESSING. Software vertex processing usage on a vertex buffer implies software vertex processing on the device. An application must create a device with software vertex processing in order to use ProcessVertices.

6.3 Transformation Matrices

Mathematically, a coordinate transformation is a mapping from one coordinate frame to another. In a single dimension, this can be thought of as converting units, such as from inches to centimeters. Inches and centimeters are related by a simple scaling transformation of 2.54 cm/in. A more general one-dimensional mapping for a quantity x is $x' = mx + b$, which allows for scaling by m and translation by b.

In three dimensions, we need to be able to rotate a point in addition to scaling and translating it. We can create a mapping similar to our one-dimensional case

with a 3x3 matrix and a 1x3 row matrix.

$$\begin{aligned} P' &= P\mathbf{M} + \mathbf{b} \\ &= \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix} \end{aligned}$$

M defines the rotation and scaling applied to P and **b** adds the translation. A vertex can also have a surface normal, which must also be transformed.

With homogeneous coordinates, we can use a single matrix for the entire transformation instead of using two matrices of different sizes. When P's cartesian coordinates are extended to homogeneous coordinates, a single 4x4 matrix can represent scaling, rotation and translation for three dimensional points. This gives $P' = P\mathbf{M}'$, with $\mathbf{M}'$ composed of elements of **M** and **b**:

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ b_1 & b_2 & b_3 & 1 \end{bmatrix}$$

The upper 3x3 submatrix corresponds to rotation and scaling and the bottom row of the matrix corresponds to translation. The homogeneous transformation matrix can also be used to implement perspective foreshortening with the rightmost column, as we will see in the next chapter. The following transformation matrices are given for a left-handed coordinate system. For rotations, the angle θ is given in radians with a counterclockwise rotation about the axis corresponding to increasing values of θ.

Translate by (x, y, z) $$\mathbf{T}(x, y, z) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x & y & z & 1 \end{bmatrix}$$

Uniform scale by s $$\mathbf{S}(s) = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Non-uniform scale by (s_x, s_y, s_z) $$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the X-axis by θ $$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the Y-axis by θ
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation about the Z-axis by θ
$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When a vertex contains a surface normal, the normal vector is specified in the same coordinate frame as the position. When the position is mapped to a new coordinate frame with a transformation matrix, the normal vector must also be mapped into the new coordinate frame. However, the normal vector shouldn't be distorted when transformed so that the proper surface orientation is preserved. If the position is transformed by $\mathbf{M}$, then transforming the normals by the inverse transpose of $\mathbf{M} = (\mathbf{M}^{-1})^T$ preserves their orientation, although they may still be scaled by the transformation.

6.4 Order of Transformations

A complex transformation, such as rotating around a point other than the origin, is built from simple transformations by multiplying the transformations together. Matrix multiplication is not commutative, so the order in which we multiply the transformation matrices is important. For instance, suppose we have two transformation matrices: $\mathbf{T}$ contains a translation and $\mathbf{R}$ contains a rotation. In general, $\mathbf{TR} \neq \mathbf{RT}$. In this case, translation followed by rotation is not the same thing as rotation followed by translation, as depicted in figure 6.1.

The simple scaling and rotation matrices operate with the origin as the center of scaling and rotation, respectively. To perform rotations about a point other than the origin, first translate the point to the origin, perform the desired rotation, and then translate the origin back to the point. Scaling about a point other than the origin is similar to rotation about a point. To rotate about an arbitrary axis, first rotate the axis to coincide with one of the three principal axes $\vec{i}$, $\vec{j}$, or $\vec{k}$, then perform the desired rotation, and then rotate the principal axis back to coincide with the axis of rotation.

Building a complex chain of transformations is a breeding ground for bugs in 3D applications. It is very easy to get one of the transformations wrong and get something that looks tantalizingly close to correct, but has anomolies. A good way to avoid mistakes is to draw a diagram of the composite transformation as a series of simple transformations. Draw one coordinate frame diagram for each simple transformation in the composite. Write down the appropriate transformation matrix for each step in the composite transformation. If the transformation can't be written as a simple transformation, break it down fur-

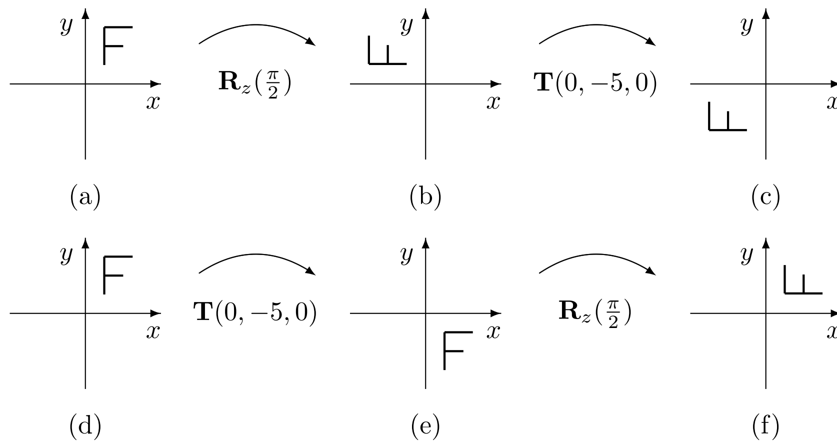


Figure 6.1: Illustration of the order of transformations on the letter "F" in the xy plane. (a)-(c) depict a rotation followed by a translation, $\mathbf{R}_z(\frac{\pi}{2})\mathbf{T}(0,-5,0)$. (d)-(f) depict the same simple transformations in the reverse order: translation followed by rotation, $\mathbf{T}(0,-5,0)\mathbf{R}_z(\frac{\pi}{2})$. The non-commutivity of matrix multiplication is seen by comparing (c) with (f).

ther until all steps are simple transformations. After all the transformations have been identified, refer to your diagram of the composite and multiply them from left to right in the order they should be applied.

Consider figure 6.2, which depicts the series of transformations needed to rotate a model of the letter "F" about the point P indicated in the figure. If we simply rotate by $\pi/2$, we end up moving the model as well as rotating it because the center of rotation is not the origin. To rotate about P, we first translate P to the origin, perform the rotation, and then transform the origin back to P. Similar composite transformations can be built for rotation about the other principal axes and for scaling.

To rotate about an arbitrary axis $\vec{a}$, you first compose three transformations that rotate $\vec{a}$ to be coincident with one of the principal axes $\vec{i}$, $\vec{j}$, or $\vec{k}$, then perform the desired rotation about that principal axis, then rotate the principal axis to be coincident with $\vec{a}$. Rotation matrices are a special case of transformation matrices in that they are commutative under multiplication, so that $\mathbf{R}_1\mathbf{R}_2 = \mathbf{R}_2\mathbf{R}_1$. Therefore, it doesn't matter in what order we multiply the three rotation matrices that rotate one axis to be coincident with another.

The D3DX library provides a collection of utility functions for generating simple transformation matrices such as $\mathbf{R}_x$. They are described in section 16.4.

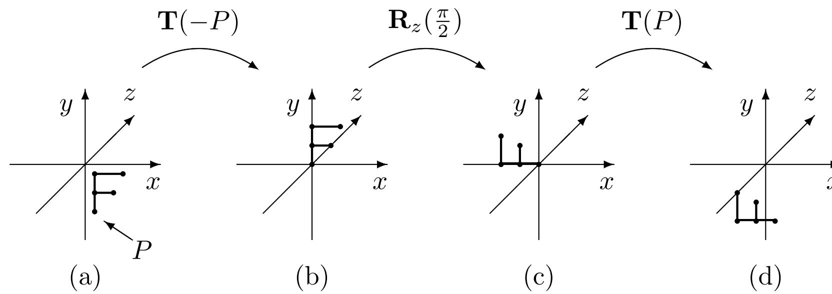


Figure 6.2: Illustration of a composite vertex transformation. The letter "F" in the xy plane is rotated clockwise by $\frac{\pi}{2}$ about the point P. The letter remains in the xy plane after the composite transformation because the axis of rotation is perpendicular to the plane.

6.5 World Transformation

In chapter 5 we described how models are built from vertices with coordinates in model space. The world transformation is the first operation applied during vertex processing and maps coordinates from model space to **world space**. The simplest transformation from model space to world space is to multiply the position component by a transformation matrix.

The surface normal component of a vertex is also specified in model space. When a vertex is transformed, we must transform not only its position component but also its normal component. While the position component is a point in space, the surface normal is a vector in space. Direct3D uses a modified transformation matrix when it transforms normals so that the orientation of the normals will be preserved. The modified matrix is the inverse transpose of the world transformation. This matrix can still contain scaling transformations, possibly changing a surface normal's length. Direct3D can compensate for changes in the length of a surface normal, see chapter 8.

The type of world transformation applied to the vertex is controlled by RS Vertex Blend, with values of type `D3DVERTEXBLENDFLAGS`. When RS Vertex Blend is `D3DVBF_DISABLE`, the vertices are processed through a single world transformation matrix.

```
typedef enum _D3DVERTEXBLENDFLAGS
{
    D3DVBF_DISABLE  = 0,
    D3DVBF_0WEIGHTS = 256,
    D3DVBF_1WEIGHTS = 1,
    D3DVBF_2WEIGHTS = 2,
    D3DVBF_3WEIGHTS = 3,
    D3DVBF_TWEENING = 255,
} D3DVERTEXBLENDFLAGS;
```

The `GetTransform` and `SetTransform` methods manipulate the transform properties of the device. The different kinds of transform properties are given by the `D3DTRANSFORMSTATETYPE`[1] enumeration and associated macros. `D3DTS_WORLD` specifies the world transformation applied when vertex blending is disabled.

```
HRESULT GetTransform(D3DTRANSFORMSTATETYPE kind,
            D3DMATRIX *value);
HRESULT SetTransform(D3DTRANSFORMSTATETYPE kind,
            const D3DMATRIX *value);

typedef enum _D3DTRANSFORMSTATETYPE {
    D3DTS_WORLD      = 256,
    D3DTS_WORLD1     = 257,
    D3DTS_WORLD2     = 258,
    D3DTS_WORLD3     = 259,
    D3DTS_VIEW       = 2,
    D3DTS_PROJECTION = 3,
    D3DTS_TEXTURE0   = 16,
    D3DTS_TEXTURE1   = 17,
    D3DTS_TEXTURE2   = 18,
    D3DTS_TEXTURE3   = 19,
    D3DTS_TEXTURE4   = 20,
    D3DTS_TEXTURE5   = 21,
    D3DTS_TEXTURE6   = 22,
    D3DTS_TEXTURE7   = 23
} D3DTRANSFORMSTATETYPE;

D3DTRANSFORMSTATETYPE D3DTS_WORLDMATRIX(UINT index);
```

6.6 Transformation Hierarchy

In addition to replacing a transformation matrix property with `SetTransform`, you can also pre-multiply a matrix onto the existing matrix with `MultiplyTransform`. If a model is composed of pieces where each piece is positioned relative to another piece, then `MultiplyTransform` can be used to compose the necessary transformations as the hierarchy is rendered.

```
HRESULT MultiplyTransform(D3DTRANSFORMSTATETYPE transform,
            const D3DMATRIX *value);
```

The diagram in figure 6.3 shows a simple robot arm consisting of two segments $L1$ and $L2$, called linkages, and locations where the segments meet another object, called joints. The entire ensemble is referred to as a jointed linkage,

[1] The world matrix symbols are shown as part of the enumeration; the header file defines them as macros with the given values.

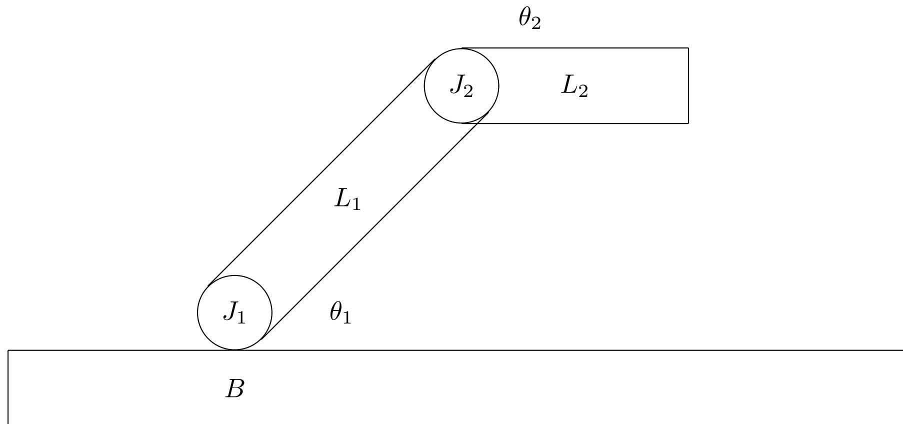


Figure 6.3: Jointed linkage transformation hierarchy showing a simple robot arm model in the xy plane consisting of a base B, two linkages L_1 and L_2, and two joints J_1 and J_2. Joint J_1 positions linkage L_1 at angle θ_1 relative to the base and joint J_2 positions linkage L_2 at angle θ_2 relative to linkage L_1. In the figure, $\theta_1 = \frac{\pi}{4}$ and $\theta_2 = -\frac{\pi}{4}$.

and can easily be drawn by creating a local coordinate frame for each piece of the model and composing transformations between local coordinate frames to position the pieces. In this simple figure, each joint has the freedom to rotate only in the xy plane.

Drawing this robot arm starts by locating the arm relative to the other objects in the scene with `SetTransform`. The base B is positioned relative to the location of the entire arm with `MultiplyTransform` and then it is drawn. Next, all the elements of the model relative to the base are drawn. The joint J_1 is relative to the base and it is positioned with `MultiplyTransform` and then drawn. This continues for linkage L_1, joint J_2 and linkage L_2 which are all relative to each other.

6.7 Vertex Blending

A jointed linkage is fine for robots, or even insects, both of which have a stiff outer shell composed of elements that move by rigid transformations. Other objects, such as cloth, plants and animals, are flexible and accurately describing their motion requires more than rigid transformations applied to pieces of the model.

One approach to simulating the deformation of the skin when an animal moves is to transform each model vertex multiple times, each with a different transformation matrix. The resulting world-space vertices are combined in a weighted average. The simplest case of vertex blending is when two matrices

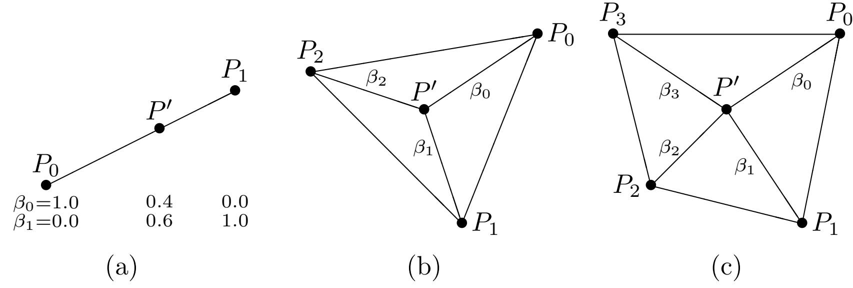


Figure 6.4: Visualization of vertex blending with two, three and four matrices. P_i are the images of the model space vertex position P under the world transformation $\mathbf{M}_i$. P' is the resulting blended vertex position. (a) β_0 moves P' along the line between P_0 and P_1. (b) β_0 and β_1 move P' inside the triangle defined by P_0, P_1 and P_2. (c) β_0, β_1 and β_2 move P' inside the region bounded by P_0, P_1, P_2, and P_3.

and a single weight is used to blend between the two transformed points. This formula is similar to the alpha blending formula presented in section 1.4. Vertex blending with a single weight may be visualized as interpolating along a straight line connecting the two transformed points P_0 and P_1, with β locating a distance along the line, as shown in figure 6.4(a).

$$\begin{aligned}
P' &= \beta_0 P_0 + (1-\beta_0)P_1 \\
&= \beta_0 P\mathbf{M}_0 + (1-\beta_0)P\mathbf{M}_1 \\
\vec{n}' &= \beta_0 \vec{n}_0 + (1-\beta_0)\vec{n}_1 \\
&= \beta_0 \vec{n}(\mathbf{M}_0^{-1})^T + (1-\beta_0)\vec{n}(\mathbf{M}_1^{-1})^T
\end{aligned}$$

Assigning a β value to each vertex defines the ratio of transformations at each vertex. Usually the blend weight values will be assigned by a modeling program where a user interface is provided for controlling the appearance of vertex blending.

The resulting appearance of a vertex blended model is influenced by the transformation matrices $\mathbf{M}_0$ and $\mathbf{M}_1$, the distribution of the weights along the model, and the model itself. Figure 6.5 plots six different distributions of blend weights on the interval $[0,1]$. Figure 6.6 shows the result of applying a translation using those distributions. The blend weights were computed for the model by normalizing the x coordinate of each vertex into the interval $[0,1]$ and computing $\beta(x)$. Figure 6.7 shows the result of applying different transformations to the same weight distribution.

We can extend this technique to using N matrices and $N-1$ blend weights per vertex instead of just two matrices. The final weight is always determined by the system to ensure that the sum of all weights is one. With two blend weights per vertex, each vertex can be positioned inside a triangle defined by

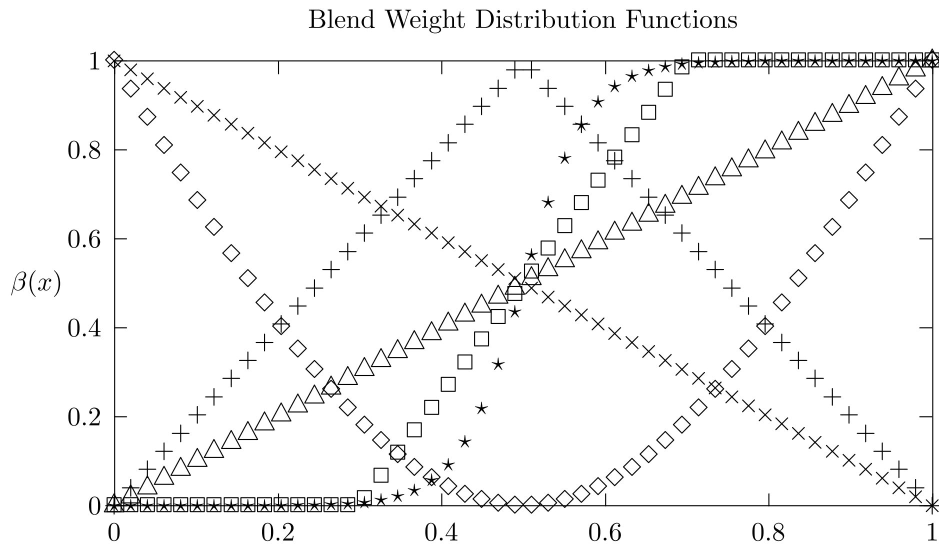


Figure 6.5: Sample blend weight functions $\beta(x)$ defined for x in $[0,1]$.

$\diamond : \beta(x) = 1 - \sin(\pi x)$

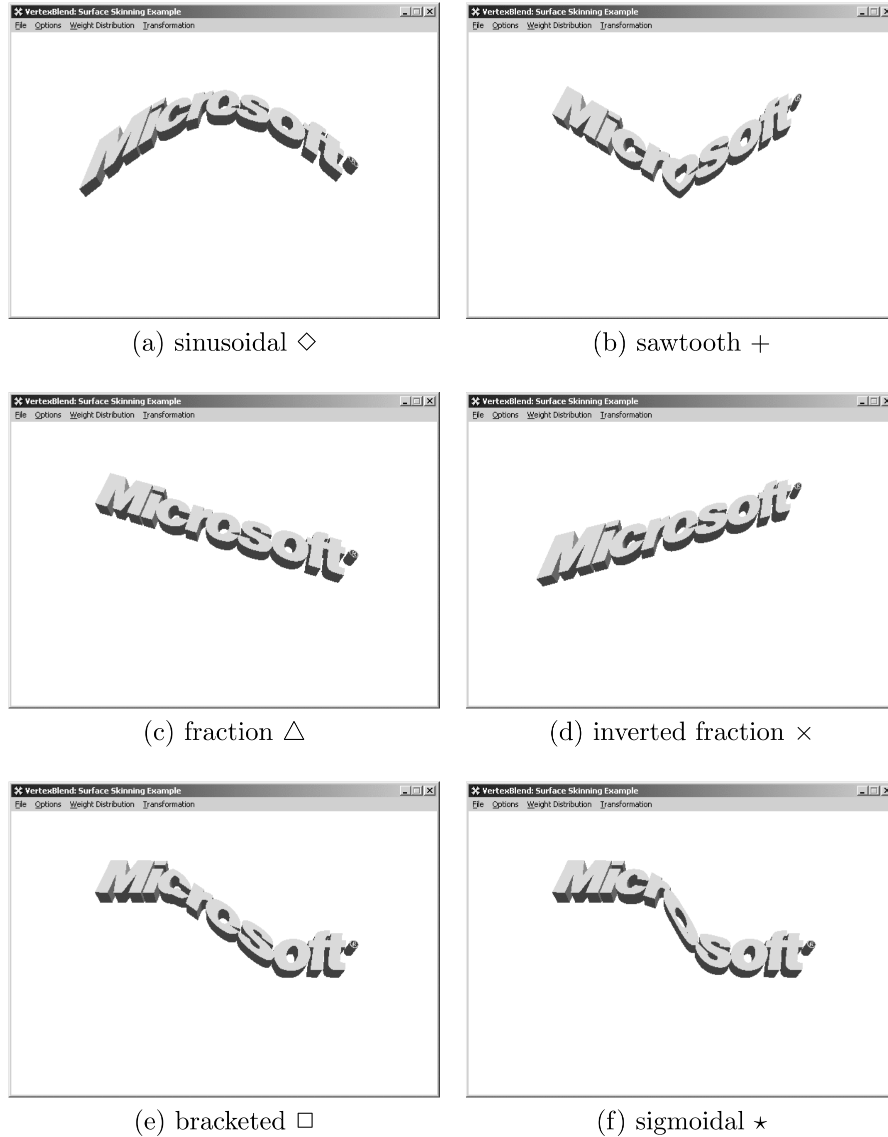
$\times : \beta(x) = 1 - x$

$\triangle : \beta(x) = x$

$\star : \beta(x) = \dfrac{1}{1 + e^{-25(x-0.5)}}$

$+ : \beta(x) = \begin{cases} 2x, \; x \in [0, 0.5] \\ 2(1-x), \; x \in (0.5, 1] \end{cases}$

$\square : \beta(x) = \begin{cases} 0, \; x \in [0, 0.3) \\ 2.5(x - 0.3), \; x \in [0.3, 0.7] \\ 1, \; x \in (0.7, 1] \end{cases}$



(a) sinusoidal ◇ (b) sawtooth +

(c) fraction △ (d) inverted fraction ×

(e) bracketed □ (f) sigmoidal ⋆

Figure 6.6: Illustration of the effect of blend weight functions on a triangle mesh of the Microsoft logo. The blend consists of a simple Y-axis translation applied to the mesh with $\mathbf{M}_0 = \mathbf{I}$ and $\mathbf{M}_1 = \mathbf{T}(0, P, 0)$. The results of applying these matrices with the different blend weight functions is shown with a suggestive name for the function and its symbol in figure 6.5.

(a) axis rotation about origin

(b) axis rotation about center

(c) scale about center

(d) Y translation

(e) Y rotation about left

(f) Y rotation about center

Figure 6.7: The effect of the world transformation on vertex blending. The same mesh is used as in figure 6.6. For all cases the blend function is the “bracketed” function □ from figure 6.5. (a) and (b) perform the same rotation about the origin and the mesh’s center point, respectively. (c) performs a uniform scaling about the center of the mesh, (d) performs a translation along the Y-axis, (e) and (f) perform the same rotation around the Y-axis about the minimum and center points of the mesh, respectively. The differences are more pronounced when the mesh is put in motion.

the three transformed points. With three weights per vertex, each vertex can be positioned inside the region bounded by the four transformed points, see figure 6.4. The following formula computes the final position P' and normal $\vec{n}'$ for $N-1$ blend weights per vertex (N matrices).

$$\begin{aligned}
\beta_N &= 1 - \sum_{k=0}^{N-1} \beta_k \\
P' &= \sum_{k=0}^{N} \beta_k P \mathbf{M}_k \\
\vec{n}' &= \sum_{k=0}^{N} \beta_k \vec{n} (\mathbf{M}_k^{-1})^T
\end{aligned}$$

6.7.1 Basic Vertex Blending

Direct3D exposes fixed-function blending through the `D3DVERTEXBLENDFLAGS` value in RS Vertex Blend. When this value is `D3DVBF_1WEIGHTS`, `D3DVBF_2-WEIGHTS`, or `D3DVBF_3WEIGHTS`, it indicates that each vertex has 1, 2, or 3 additional floats in the position component. The floats give the β_0, β_1 and β_2 blend weights for each vertex used with 2, 3, or 4 matrices, respectively. If `D3DVBF_-0WEIGHTS` is specified, a single matrix with a weight of 1.0 for each vertex is used. This gives the same result as `D3DVBF_DISABLE`. RS Vertex Blend can be set to use fewer weights than are supplied with the vertex. The final weight will still be computed so that all the blend weights sum to one. Setting RS Vertex Blend to use more weights than are defined in the vertex is an error. The number of matrices that can be used at any one time is given by `D3DCAPS9-::MaxVertexBlendMatrices`, which may vary based on the device's creation parameters.

When using an FVF to describe vertices, the `D3DFVF_XYZB`n FVF flags specify the number of blend weights stored in the position component of each vertex. With `D3DFVF_XYZB1`, `D3DFVF_XYZB2` and `D3DFVF_XYZB3` the final blend weight is computed implicitly from the blend weights given in the vertices. With `D3D-FVF_XYZB4` the runtime uses β_0, β_1, β_2, and β_3 as the blend weights directly without computing the fourth weight implicitly. With `D3DFVF_XYZB5` the runtime uses β_0, β_1, β_2, β_3 as the explicit blend weights and β_4 as the blend matrix indices for indexed vertex blending.

When using a vertex shader declaration to describe vertices, the blend weights are mapped to the blend weights usage semantic. The blend matrix indices are mapped to the blend indices usage semantic.

The blend weights β_0, β_1, β_2 and β_3 are associated with the four blend matrices `D3DTS_WORLD`, `D3DTS_WORLD1`, `D3DTS_WORLD2`, or `D3DTS_WORLD3`. The same matrices can be specified with the `D3DTS_WORLDMATRIX()` macro using the arguments 0, 1, 2 or 3, respectively. The blend matrices are manipulated with

calls to GetTransform and SetTransform.

In Direct3D, the blend matrix includes the world transformation **W** for the vertex as well as its deforming transformation **D**. Matrix composition can be used when the deforming transformation is given in world coordinates, or when an additional world transformation is to be applied after a model space deformation. In the former, use **WD** for the vertex blend matrix to map the model to world coordinates and then perform a deformation in world coordinates. In the latter, use **DW** for the vertex blend matrix to perform the deformation in model space and then map the blended vertex to world coordinates.

6.7.2 Indexed Vertex Blending

While a broad range of deformations can be created with vertex blending, it is still somewhat restrictive for a complex model with many joints, like a human being. With vertex blending the maximum number of transformations that can be applied by the API is four per vertex (or triangle) and the same transformations must be used for all vertices in each call to the DrawPrimitive methods.

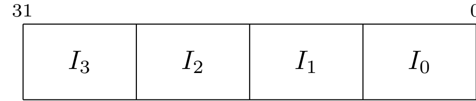
With indexed vertex blending, also called matrix palette skinning, matrix indices are stored in each vertex and each index selects a matrix from a palette to use for each weight. A vertex with N blend weights will use $N+1$ matrices and have an additonal $N+1$ matrix indices. This allows up to four independent matrices per vertex or twelve matrices per triangle. The blend formulas for indexed vertex blending are identical to the vertex formulas with an additional level of indirection through the blend matrix indices I_k.

$$\begin{aligned}
\beta_N &= 1 - \sum_{k=0}^{N-1} \beta_k \\
P' &= \sum_{k=0}^{N} \beta_k P \mathbf{M}_{I_k} \\
\vec{n}' &= \sum_{k=0}^{N} \beta_k \vec{n} (\mathbf{M}_{I_k}^{-1})^T
\end{aligned}$$

Indexed vertex blending is enabled through RS Indexed Vertex Blend Enable, and vertex blending is controlled through D3DRS_VERTEXBLEND. The size of the matrix palette for a device is given by D3DCAPS9::MaxVertexBlendMatrix-Index. If this value is zero, then the device does not support indexed vertex blending. The number of supported indices per vertex is the same as the number of supported vertex blend matrices.

With an FVF vertex buffer, the indices are specified by adding an additional "weight" to each vertex and including the flag D3DFVF_LASTBETA_UBYTE4. This flag reinterprets the last vertex blend weight in the vertex as a DWORD containing

the matrix indices. The indices are stored as `BYTE`s packed into a single `DWORD`, giving a range of $[0, 255]$ for blend matrix indices. I_0, the index associated with β_0 and $\mathbf{M}_0$, is stored in the least significant `BYTE` of the `DWORD`. I_1 is stored in the next signficant `BYTE`, and so-on.



TODO: Work out UBYTE4 presence

With a fixed-function vertex shader, the matrix index stream data is mapped to the blend indices usage semantic. With a programmable vertex shader, the stream data can be mapped to any input register. As described in section 5.8, the `D3DDECLTYPE_UBYTE4` stream data type may not be available. In this situation, the indices are declared as `D3DDECLTYPE_D3DCOLOR`, but they will need to be rescaled from the $[0, 1]$ range, see chapter 9. If the blending indices are declared as `D3DDECLTYPE_SHORT2` or `D3DDECLTYPE_SHORT4`, then no scaling is required.

6.7.3 Vertex Tweening

Some vertex blending effects can't be achieved with vertex blending through matrices, but can be achived with a method called **tweening**.[2] The name derives from traditional film animation where a character is drawn with a beginning and ending pose by one animator and another animator drew the "in between" frames that move the character from one pose to another.

The fixed-function pipeline provides vertex tweening in a similar fashion to the approach used in film animation. A device supports Vertex tweening when the `D3DVTXPCAPS_TWEENING` bit of `D3DCAPS9::VertexProcessingCaps` is set.

```
#define D3DVTXPCAPS_TWEENING 0x00000040L
```

Each vertex is defined with two positions, and optionally two normals. The two components P_1 and P_2 are combined with a tween factor f before being transformed by a the world transformation matrix $\mathbf{M}$.

$$\begin{aligned} P' &= [(1-f)P_1 + fP_2]\mathbf{M} \\ \vec{n}' &= [(1-f)\vec{n}_1 + f\vec{n}_2](\mathbf{M}^{-1})^T \end{aligned}$$

Vertex tweening is enabled when RS Vertex Blend is `D3DVBF_TWEENING`. The tween factor f is supplied by RS Tween Factor and must be in the range $[0, 1]$. $\mathbf{M}$ is the `D3DTS_WORLD` matrix.

Fixed-function vertex tweening must be used with a vertex shader declaration; it cannot be used with an FVF code shader. Tweening also requires

[2] Also referred to as "morphing", from the word metamorphosis.

that the vertex components for the second position and normal appear last in the vertex. The vertex declaration maps the stream data for P_1 and P_2 to the position usage semantic for usage indices zero and one, respectively. If normals are present, both n_1 and n_2 must be present. n_1 and n_2 are mapped to normal usage semantic for usage indices zero and one, respectively.

With vertex tweening, each vertex moves independently of all other vertices, while with vertex blending, they all move under the influence of the same set of transformations. While vertex tweening cannot introduce or remove vertices, vertices can be made coincident to produce degenerate primitives that are not rasterized.

6.7.4 Blending Within Device Limitations

Vertex blending is a powerful technique, but may not be supported natively on the hardware. The device may not have hardware vertex processing, or it might not support more than a single world transformation matrix, or it might not support indexed vertex blending, or it might support too few matrices in the matrix palette.

However, even a model with many joints and vertex blending transformations doesn't use all the transformations for every triangle. Usually it is possible to partition the model into groups of primitives, where all primitives in a group use at most the number of blend matrices supported on the device. For each group, the blend matrices are set appropriately.

If a group of primitives uses more blend matrices than are supported in hardware, you can eliminate the transformation with the smallest contribution to the final blend and renormalize the remaining weights. For every primitive in the group find the blend transformation k that contributes least to all the primitives in the group. Remove β_k from each vertex for each primitive in the group, and compute new blend weights by renormalizing the remaining weights.

$$\beta_i' = \frac{\beta_i}{1-\beta_k}$$

Another approach to device vertex blending limitations is to change the way vertex blending is performed. Software vertex processing fully supports vertex blending and can be used in cases where hardware vertex processing is insufficient. A programmable vertex shader can perfrom vertex blending of arbitrary complexity, limited only by vertex shader capabilities and performance. The runtime uses CPU optimized code for software vertex processing of both fixed-function and programmable vertex shaders, making this a viable alternative for not much effort.

When all else fails, an application can simply disable vertex blending when support is lacking, using only rigid transformations for orienting models.

D3DX contains functions for converting an arbitrary skinned mesh into a mesh using vertex blending or indexed vertex blending, see chapter 19.

6.8 Vertex Fog

Fog, also called depth-cueing, is an effect that changes an object's color based on its depth from the camera. The object's color C is changed by blending it with the fog color C_f using a fog factor f.

$$C' = fC + (1 - f)C_f$$

The fog factor f is computed based on the object's distance from the camera. Fog changes the color of objects, but it does not change an object's transparency, so the alpha channel remains untouched by the fog blend.

Direct3D provides support for two kinds of fog application: **vertex fog** and **pixel fog** Only one of these can be used at a time and pixel fog cannot be used with programmable vertex shaders. With vertex fog, the fog factors are computed per-vertex as part of vertex processing and these fog factors are interpolated for each pixel produced by rasterization. With pixel fog, also called **table fog**, the fog factors are computed per-pixel during rasterization, usually by a table lookup from the pixel's depth. Direct3D also allows an application to compute the fog factor for each vertex and supply it to the rasterizer for interpolation. Once the fog factor has been determined, the fog blend is applied to the pixel as the last stage of pixel processing.

RS Fog Enable controls the application of fog altogether. If this state is `FALSE`, then no fog computations are performed. The fog color C_f is defined by RS Fog Color; only the RGB channels are used. RS Fog Vertex Mode and RS Fog Table Mode contain values of `D3DFOGMODE` and select either vertex or pixel fog, respectively. A device supports vertex or pixel fog on triangle primitives if the `D3DPRASTERCAPS_FOGVERTEX` or `D3DPRASTERCAPS_FOGTABLE` bit of `D3DCAPS9::RasterCaps` is set, respectively. If the `D3DLINECAPS_FOG` bit of `D3DCAPS9::LineCaps` is set, then fog is supported on point and line primitives.

```
typedef enum _D3DFOGMODE {
    D3DFOG_NONE   = 0,
    D3DFOG_LINEAR = 3,
    D3DFOG_EXP    = 1,
    D3DFOG_EXP2   = 2
} D3DFOGMODE;
```

With fixed-function processing, one or both of RS Fog Table Mode and RS Fog Vertex Mode can be set to `D3DFOG_NONE`. Enabling RS Fog Table Mode and RS Fog Vertex Mode simultaneously results in an error. To supply application computed fog factors, either through stream data or a programmable vertex shader, set both fog modes to `D3DFOG_NONE` and RS Fog Enable to `TRUE`. The fog mode selects a formula that is used to compute the fog factor based on depth. The formulas are plotted in figure 6.8 and given below.

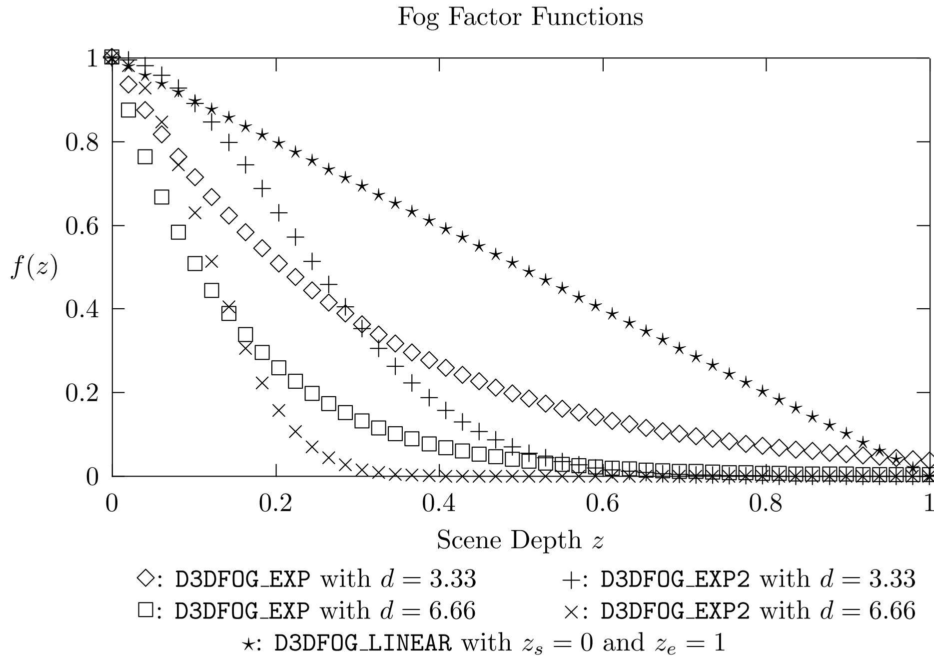


Figure 6.8: Fog factor functions $f(z)$ plotted versus scene depth z.

$$\texttt{D3DFOG_LINEAR} \quad f(z) = \begin{cases} 1, \; z < z_s \\ \frac{z_e - z}{z_e - z_s}, \; z \in [z_s, z_e] \\ 0, \; z > z_e \end{cases}$$

$$\texttt{D3DFOG_EXP} \quad f(z) = e^{-dz}$$

$$\texttt{D3DFOG_EXP2} \quad f(z) = e^{-(dz)^2}$$

Linear fog ramps from the object's color to the fog color when the depth is in the range $[z_s, z_e]$. RS Fog Start and RS Fog End specify z_s and z_e, respectively. Exponential fog makes a smoother transition between the object's color and the fog color. Exponential fog has a density parameter d that controls the falloff of the fog factor; d is defined by RS Fog Density.

The fog distance is usually computed as the distance to a plane at depth z from the camera. For points distant from the center of the image, this is not the true distance between the camera and the point. Range-based fog computes the true distance from a point to the camera for use with the fog factor formula. If the `D3DPRASTERCAPS_FOGRANGE` bit of `D3DCAPS9::RasterCaps` is set, the device supports range-based fog. Range-based fog is controlled with RS Range Fog Enable.

The coordinate space of the depth value used in fog computations can vary. With vertex fog, the distance is measured in camera space before projection. The range of depth values is $[z_n, z_f]$ where z_n and z_f are the near and far planes

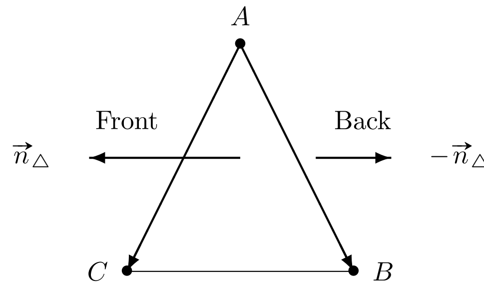


Figure 6.9: Computing the plane normal for triangle ABC with the vertices $\{A,\ B,\ C\}$. The two vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$ are independent vectors in the plane of the triangle. The cross-product of these two vectors is perpendicular to the plane containing the triangle.

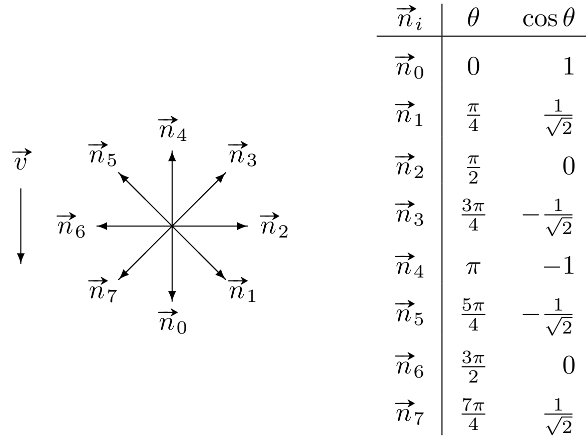
of the view volume, as described in chapter 7. With pixel fog, the distance is measured either in the unit interval $[0, 1]$ with Z buffering or in camera space with W buffering. W buffering is an alternative to Z buffering. Both are discussed in chapter 14. The `D3DPRASTERCAPS_ZFOG` and `D3DPRASTERCAPS_WFOG` bits of `D3DCAPS9::RasterCaps` indicate that the device supports Z buffer fog and W buffer fog, respectively.

With fixed-function vertex processing, application supplied fog factors are stored in the alpha channel of the specular color vertex component. With programmable vertex processing, the application is free to store arbitrary per-vertex fog data in the data stream, or generate it entirely within the shader from other vertex data such as position, see chapter 9.

6.9 Face Culling

Most models have about half of their triangles facing the viewer. When the triangle is positioned relative to the viewer, the triangle's normal vector points towards the viewer if the triangle is visible. The surface normal points away from the viewer when the triangle faces away from the viewer. Triangles facing away from the viewer are "back-facing" triangles. We can use the relative orientation of the triangle normal to **cull** triangles from further processing. This avoids all the rasterization processing for triangles that won't be seen.

Face culling uses the normal to the plane containing the triangle. Direct3D does not use the surface normal vertex component for culling. Face culling is a property of the triangle, not the smooth surface approximated by the triangle. The normal is computed from the triangle's vertices. Any plane normal vector is the cross-product of two linearly independent vectors in the plane. "Linearly independent" just means that the two vectors shouldn't point in the same direction, or exactly opposite to each other. The triangle ABC in figure 6.9 has vertices $A(x_a, y_a, z_a)$, $B(x_b, y_b, z_b)$ and $C(x_c, y_c, z_c)$, in clockwise order. The vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$ satisfy the cross-product condition. Their cross-product



$\vec{n}_i$	θ	$\cos\theta$
$\vec{n}_0$	0	1
$\vec{n}_1$	$\frac{\pi}{4}$	$\frac{1}{\sqrt{2}}$
$\vec{n}_2$	$\frac{\pi}{2}$	0
$\vec{n}_3$	$\frac{3\pi}{4}$	$-\frac{1}{\sqrt{2}}$
$\vec{n}_4$	π	-1
$\vec{n}_5$	$\frac{5\pi}{4}$	$-\frac{1}{\sqrt{2}}$
$\vec{n}_6$	$\frac{3\pi}{2}$	0
$\vec{n}_7$	$\frac{7\pi}{4}$	$\frac{1}{\sqrt{2}}$

Figure 6.10: Determining back-facing triangles for triangle normal vectors $\vec{n}_i$ and the line of sight vector $\vec{v}$. The angle θ between the two vectors is computed from their dot product $\vec{n}_i \cdot \vec{v}$. The normals $\vec{n}_3$, $\vec{n}_4$, and $\vec{n}_5$ are back-facing. The normals $\vec{n}_2$ and $\vec{n}_6$ are edge-on to the viewer, resulting in a degenerate triangle. The normals $\vec{n}_0$, $\vec{n}_1$ and $\vec{n}_7$ are front-facing.

gives the plane normal $\vec{n}_\triangle$.

$$
\begin{aligned}
\overrightarrow{AB} &= \langle x_b - x_a,\, y_b - y_a,\, z_b - z_a \rangle \\
\overrightarrow{AC} &= \langle x_c - x_a,\, y_c - y_a,\, z_c - z_a \rangle \\
\vec{n}_\triangle &= \overrightarrow{AB} \otimes \overrightarrow{AC}
\end{aligned}
$$

The order of the vertices defining the triangle can reverse the sign of the resulting normal vector. The order of the vectors in the cross-product operation also reverses the sign of the normal vector since $\vec{a} \otimes \vec{b} = -\vec{b} \otimes \vec{a}$. The formulas given above are for the triangle with vertices specified in a clockwise order.

Once the plane normal for the triangle has been determined based on the winding order, the relative orientation of the plane normal and the line of sight of the camera can be computed with a vector dot product. Figure 6.10 illustrates the relative positioning of the plane normal $\vec{n}$ to the line of sight vector $\vec{v}$. If the sign of the dot product is negative, then the triangle is back-facing.

After vertices have been transformed, the line of sight is straight down the z axis. In this situation, the orientation of a triangle is simplified into determining the winding order of the transformed vertices. In figure 6.9 the transformed vertices $\{A, B, C\}$ are in clockwise order and represent a front-facing triangle. The triangle with transformed vertices $\{A, C, B\}$ are in counter-clockwise order and represent a back-facing triangle.

Direct3D assumes that all visible triangles are front-facing. Face culling operates best on models without holes, where no back-facing triangles are visible.

If the exterior surface of an object is modeled with a "hole", the interior of the model will be visible. If back facing triangles are culled, then the interior will not be visible at all. Even when the triangles are not culled, they are improperly processed by Direct3D because their surface normals point in the wrong direction for the interior surface. The solution is to also model the interior of the object. If clipping planes are used to cut away an object, capping geometry can be computed at the clip plane intersection to avoid back-facing triangles.

Face culling is specified by RS Cull Mode of type `D3DCULL`. `D3DCULL_NONE` disables face culling. `D3DCULL_CW` culls triangles whose transformed vertices form a clockwise path. `D3DCULL_CCW`, the default, culls triangles whose transformed vertices form a clockwise path. Lines 43–48 of listing 2.1 on page 45 lists vertices for a single screen-space triangle in clockwise order. If lines 46 and 47 are reversed, the screen-space orientation of the triangle will be counter-clockwise and the triangle will be culled.

```
typedef enum _D3DCULL {
    D3DCULL_NONE = 1,
    D3DCULL_CW   = 2,
    D3DCULL_CCW  = 3
} D3DCULL;
```

A device indicates face culling support with three bits of the `PrimitiveMiscCaps` member of `D3DCAPS9`. Each set bit indicates support for the corresponding cull mode.

```
#define D3DPMISCCAPS_CULLNONE 0x00000010L
#define D3DPMISCCAPS_CULLCW   0x00000020L
#define D3DPMISCCAPS_CULLCCW  0x00000040L
```

6.10 Clipping

Primitives are clipped when they cross a boundary. The boundary can be the edge of the render target, or it can be an arbtirary plane in 3-space. When a primitive is clipped, Direct3D determines the portion inside the boundary and renders that portion. The rendered portion is determined by computing the intersection of the primitive with the boundary plane, introducing new vertices at the intersection.

The new vertices are computed by finding the intersection of the geometry with the clip plane, see figure 6.11. Points are clipped when the center of the point is outside the clip plane. In the figure, point B is clipped. Points A and C are not clipped. Lines are clipped by determining the intersection point and rendering the inside portion. Line segment AB is clipped to produce the line segment AB'. Line segment AC is not clipped. A triangle is clipped by determining the intersection of the clip plane with the sides of the triangle. This may require the introduction of an additional vertex for a clipped triangle. Triangle ABC is clipped to produce two triangles $AB'B''$ and $AB''C$.

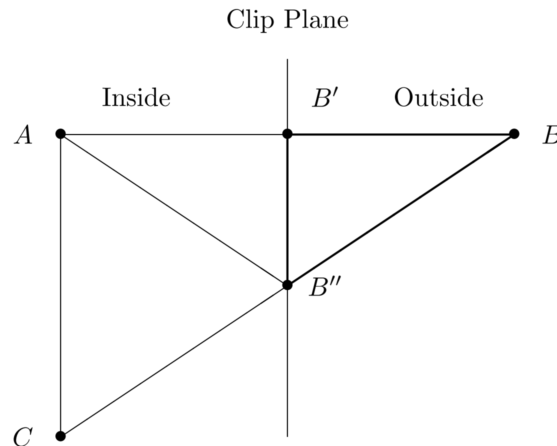


Figure 6.11: An illustration of clipping. The clipping plane is shown edge-on and the depicted geometry straddles the clip plane. The geometry intersects the clip plane at the points B' and B''. The geometry shown with bold lines is clipped and removed.

When lines and triangles are clipped, new vertices are produced. Surface normals, texture coordinates, diffuse and specular colors are interpolated between the two vertices along the intersected segment. In figure 6.11, the component data for B'' is a linear combination of the component data for B and C.

Direct3D provides several styles of clipping. User defined clip planes can be specified by the application to clip geometry against arbitrary planes. Frustum clipping is applied when the view volume is constructed. The view volume is the volume of space that is shown by the camera and is described in chapter 7. Guard band clipping is similar to view volume clipping and describes a performance optimization available on some devices.

Some of the frame buffer operations, such as Z test, alpha test, stencil test and some pixel shader operations can also be thought of as "clipping" operations. However, these operate on the pixels produced by rasterizing a primitive, and not on the geometry.

For sophisticated clipping requirements, an application can apply clipping itself to vertices through the same intersection and linear interpolation used by Direct3D.

When clipping is not enabled and primitives are drawn straddling the border, then the entire primitive (point, line or triangle) is not drawn.

6.10.1 User Clip Planes

User-defined clip planes provide up to 6 arbitrary planes that can be used to clip input geometry. Each plane is defined by a set of coefficients in a coordinate system. User defined clip planes can be used with the fixed-function pipeline

and with programmable vertex processing. With fixed-function processing, the plane's coefficients are specified in world coordinates. For the programmable vertex processing, the plane's coefficients are specified in the output space of vertices, see chapter 9.

The number of clip planes supported by a device is given by `D3DCAPS9::-MaxUserClipPlanes`. Each clip plane is defined with four coefficients.

$$ax + by + cz + d = 0$$

The normal vector to the plane is the vector $\langle a, b, c \rangle$. This vector is not necessarily a unit vector. The coefficient d fixes the plane to intersect with the point (x_0, y_0, z_0).

$$d = -(ax_0 + by_0 + cd_0)$$

The clip planes are numbered 0 through 5. Their coefficients are stored as an array of `float`s in the order a, b, c, d. The clip planes are manipulated with `GetClipPlane` and `SetClipPlane`.

```
HRESULT GetClipPlane(DWORD index,
            float *value);
HRESULT SetClipPlane(DWORD index,
            const float *value);
```

Each user-defined clip plane can be independently enabled or disabled with RS Clip Plane Enable. The ith clip plane is enabled when bit 2^i of RS Clip Plane Enable is set. Macros are provided for computing the bits.

```
#define D3DCLIPPLANE0 (1 << 0)
#define D3DCLIPPLANE1 (1 << 1)
#define D3DCLIPPLANE2 (1 << 2)
#define D3DCLIPPLANE3 (1 << 3)
#define D3DCLIPPLANE4 (1 << 4)
#define D3DCLIPPLANE5 (1 << 5)
```

D3DX provides a plane class that can be used for manipulating planes, see chapter 16.

6.10.2 View Frustum Clipping

Primitives can be clipped against the view frustum when rendered. If RS Clipping is `TRUE`, then clipping is enabled. On some devices, clipping can add a considerable overhead. If the scene is entirely within the view frustum and does not require clipping, then RS Clipping can be set to `FALSE` to avoid any clipping computations.

When view frustum clipping or user defined clipping is enabled, the device returns information about clipped primitives in the clip status property. While primitives are processed, any primitive that intersects an enabled clipping plane will set a bit in the clip status for the intersected plane. If a primitive is

completely outside the view frustum, it does not intersect the view frustum and is culled.

The clip status is manipulated with the `GetClipStatus` and `SetClipStatus` methods. The clip status itself is stored in a `D3DCLIPSTATUS9` structure.

```
HRESULT GetClipStatus(D3DCLIPSTATUS9 *value);
HRESULT SetClipStatus(const D3DCLIPSTATUS9 *value);

typedef struct _D3DCLIPSTATUS9
{
    DWORD ClipUnion;
    DWORD ClipIntersection;
} D3DCLIPSTATUS9;
```

The clip status accumulates information about clipping until it is reset. The `ClipUnion` member has a bit set for any enabled clip plane that clipped all vertices. The `ClipIntersection` member has a bit set for each clip plane that clipped all primitives. The `D3DDDDCS` flags define individual bits for each of the clip planes.

```
#define D3DCS_LEFT   0x00000001L
#define D3DCS_RIGHT  0x00000002L
#define D3DCS_TOP    0x00000004L
#define D3DCS_BOTTOM 0x00000008L
#define D3DCS_FRONT  0x00000010L
#define D3DCS_BACK   0x00000020L
#define D3DCS_PLANE0 0x00000040L
#define D3DCS_PLANE1 0x00000080L
#define D3DCS_PLANE2 0x00000100L
#define D3DCS_PLANE3 0x00000200L
#define D3DCS_PLANE4 0x00000400L
#define D3DCS_PLANE5 0x00000800L
```

6.10.3 Guard Band Clipping

Guard band clipping is similar to view frustum clipping, but it operates entirely on pixels and not on model geometry. The guard band is an area enclosing the current rendering viewport. Any primitives rendered outside the current viewport but within the guard band will have their pixels discarded.

The guard band allows expensive geometry clipping tests and interpolation to be skipped if it can be guaranteed that all geometry lies within the guard band. If rendered geometry extends beyond the guard band, then view frustum clipping must be used to clip the primitives.

The extent of the guard band is given by the `GuardBandLeft`, `GuardBand-Right`, `GuardBandTop` and `GuardBandBottom` members of `D3DCAPS9`. The guard band is given in screen space coordinates. To determine if a model is within the guard band, the comparison must be made in the same coordinate frame.

Either the model must be mapped to screen space, or the guard band must be mapped to world space.

6.11 Screen Space and the Viewport

The homogeneous divide and viewport application lie between the vertex and screen space. After the world, view and projection transforms have been applied, the vertices are homogeneous points (x, y, z, w) within the **canonical view volume**:

$$\begin{array}{rcccl} -w & \leq & x & \leq & w \\ -w & \leq & y & \leq & w \\ 0 & \leq & z & \leq & w \end{array}$$

Dividing all the coordinates of the view volume by w gives a canonical view volume in cartesian coordinates:

$$\begin{array}{rcccl} -1 & \leq & x/w & \leq & 1 \\ -1 & \leq & y/w & \leq & 1 \\ 0 & \leq & z/w & \leq & 1 \end{array}$$

This produces the $1/w$ reciprocal homogeneous w term present in transformed vertices with `D3DFVF_XYZRHW`.

With the vertices in a normalized cartesian coordinate system, they are ready for the final mapping into screen space. This mapping is defined by the viewport. The viewport can restrict rendering to a subrectangle of the render target, to a subinterval of the depth buffer, or both. The viewport is defined by a rectangle in screen space that limits the xy extent of rendering and an interval in $[0, 1]$ that limits the extent of rendering in depth. The default viewport covers the entire render target and the entire range of depth values. When the render target is changed, the viewport is reset to cover the entire extent of the new render target and the entire range of depth values.

The viewport mapping can also be written as a composite transformation matrix:

$$\mathbf{M} = \mathbf{S}\left(\frac{w}{2}, \frac{h}{2}, z_f - z_n\right)\mathbf{T}(x_s, y_s, z_n)$$

This transformation maps x from $[0, 1]$ to $[x_s, x_s + w]$, y from $[0, 1]$ to $[y_s, y_s + h]$, and z from $[0, 1]$ to $[z_n, z_f]$. Only the position vertex component is affected by the viewport.

The viewport is manipulated with the `GetViewport` and `SetViewport` methods. The viewport itself is stored in a `D3DVIEWPORT9` structure. The xy screen extent is given by a rectangle with its upper left given by the `X` and `Y` members. The `Width` and `Height` members give the dimensions of the viewport rectangle, in pixels. The `MinZ` and `MaxZ` members give the portion of the depth buffer that will be used.

```
HRESULT GetViewport(D3DVIEWPORT9 *value);
HRESULT SetViewport(const D3DVIEWPORT9 *value);
```

```
typedef struct _D3DVIEWPORT9
{
    DWORD X;
    DWORD Y;
    DWORD Width;
    DWORD Height;
    float MinZ;
    float MaxZ;
} D3DVIEWPORT9;
```

The range of z values allows an application to perform some visibility effects with the depth buffer. For instance, the depth buffer can be cleared to 1.0, the scene can be rendered in the range $[0, 0.9]$ and a background can be rendered in the range $[0.9, 1.0]$. This ensures that the background is behind all foreground elements, but elements within the background still have a range of depth values to use for occlusion among themselves.

Once the viewport has been applied, vertex processing is complete. The screen-space vertices are then interpolated during rasterization to produce a stream of pixels for integration into the render target.

6.12 rt_VertexBlend Sample Application

The sample application is a modified version of the vertex blending sample in the SDK. This sample was used to create figure 6.6 and figure 6.7. `rt_VertexBlend` was generated with the SDK framework using the AppWizard included with the SDK, see appendix A.

The entire source code is included in the samples accompanying this book. Listed here is `rt_VertexBlend.cpp`, containing the "interesting" code of the sample. Lines 50–77 check device capabilities for the skinning methods used by the sample. Lines 257–323 compute the blend world matrices. An orienting transformation is post-multiplied onto the blend matrices in lines 327–330. The user can change the orienting transformation with the arrow keys on the keyboard. The blending weight for each vertex is computed in lines 440–474. The x coordinate of the position component is normalized across the x extent of the model and used as the input to a blend weight distribution function.

The sample uses small helper classes that encapsulate reusable Direct3D coding idioms. Their meaning should be straightforward and all such helper classes are placed in the `rt` namespace to highlight their use. The `mat` classes construct transformation matrices so that matrix expressions can be more readily composed. The sample source code contains their definitions.

Listing 6.1: `rt_VertexBlend.cpp`: Vertex blending.

```
1   ////////////////////////////////////////////////////////////
2   // rt_VertexBlend.cpp
```

```
3   //
4   // A modification of the VertexBlend sample to demonstrate
5   // more clearly the effect of the blend weight distribution
6   // and transformation matrices on the result of vertex
7   // blending.
8   //
9
10  // C++ includes
11  #include <cmath>
12
13  // Win32 includes
14  #define STRICT
15  #define WIN32_LEAN_AND_MEAN
16  #include <windows.h>
17
18  // ATL includes
19  #include <atlbase.h>
20
21  // Direct3D includes
22  #include <d3dx9.h>
23
24  // D3DFrame includes
25  #include "DXUtil.h"
26  #include "D3DEnumeration.h"
27  #include "D3DSettings.h"
28  #include "D3DApp.h"
29  #include "D3DFont.h"
30  #include "D3DFile.h"
31  #include "D3DUtil.h"
32
33  // sample includes
34  #include "resource.h"
35  #include "rt_VertexBlend.h"
36
37  // rt includes
38  #include "rt/app.h"
39  #include "rt/hr.h"
40  #include "rt/mat.h"
41  #include "rt/media.h"
42  #include "rt/mesh.h"
43  #include "rt/misc.h"
44  #include "rt/states.h"
45  #include "rt/tstring.h"
46
47  ///////////////////////////////////////////////////////////
48  // s_blend_vertex
```

```
49  //
50  // Vertex structure that includes a blend weight.
51  //
52  struct s_blend_vertex       // Referenced in the shader as:
53  {
54      D3DXVECTOR3 v;          // v0
55      float       blend;      // v1.x
56      D3DXVECTOR3 n;          // v3
57      float       tu, tv;     // v7
58
59      static const DWORD FVF;
60  };
61  const DWORD s_blend_vertex::FVF =
62      D3DFVF_XYZB1 | D3DFVF_NORMAL | D3DFVF_TEX1 | D3DFVF_TEXCOORDSIZE2(0);
63
64  ////////////////////////////////////////////////////////////
65  // ConfirmDevice()
66  //
67  // Validate both vertex blending and vertex shader caps
68  // and set available application options accordingly.
69  // Check that the device supports at least one of the
70  // two techniques used in this sample: either a vertex
71  // shader, or at least two blend matrices and a
72  // directional light.
73  //
74  HRESULT
75  CMyD3DApplication::ConfirmDevice(D3DCAPS9 *caps,
76      DWORD dwBehavior, D3DFORMAT)
77  {
78      if (dwBehavior & D3DCREATE_SOFTWARE_VERTEXPROCESSING)
79      {
80          // Software vertex processing always supports what
81          // we need
82          return S_OK;
83      }
84
85      if (caps->VertexShaderVersion >= D3DVS_VERSION(1,0))
86      {
87          // HW device supports a vertex shader
88          return S_OK;
89      }
90
91      // Fixed-function HAL:
92      // Check for vertex blending with at least two matrices
93      // (Software can always do up to 4 blend matrices)
94      if (caps->MaxVertexBlendMatrices < 2)
```

```
 95     {
 96         return E_FAIL;
 97     }
 98
 99     // check for directional lights
100     if (!(caps->VertexProcessingCaps &
101             D3DVTXPCAPS_DIRECTIONALLIGHTS))
102     {
103         return E_FAIL;
104     }
105
106     return S_OK;
107 }
108
109 ////////////////////////////////////////////////////////////
110 // InitDeviceObjects()
111 //
112 // Load the mesh and adjust behavior of app to follow caps
113 // of the device.
114 //
115 HRESULT CMyD3DApplication::InitDeviceObjects()
116 {
117     // Init the font
118     THR(m_font.InitDeviceObjects(m_pd3dDevice));
119
120     // Load an object to render
121     if (FAILED(m_mesh.Create(m_pd3dDevice,
122         _T("mslogo.x"))))
123     {
124         return D3DAPPERR_MEDIANOTFOUND;
125     }
126     // Set a custom FVF for the mesh
127     m_mesh.SetFVF(m_pd3dDevice, s_blend_vertex::FVF);
128
129     // compute bounding box
130     {
131         CComPtr<ID3DXMesh> mesh = m_mesh.GetSysMemMesh();
132         rt::mesh_vertex_lock<D3DXVECTOR3> lock(mesh);
133         THR(::D3DXComputeBoundingBox(lock.data(),
134             mesh->GetNumVertices(), sizeof(s_blend_vertex),
135             &m_mesh_bounds.m_min, &m_mesh_bounds.m_max));
136     }
137
138     // adjust behavior to follow caps
139     if ((m_dwCreateFlags &
140             (D3DCREATE_HARDWARE_VERTEXPROCESSING |
```

```
141                  D3DCREATE_MIXED_VERTEXPROCESSING)) &&
142             m_d3dCaps.VertexShaderVersion < D3DVS_VERSION(1,0))
143         {
144             // No vertex shaders
145             m_use_vertex_shader = false;
146             rt::enable_menu(::GetMenu(m_hWnd),
147                 ID_OPTIONS_USECUSTOMVERTEXSHADER, false);
148         }
149         else if (m_d3dCaps.MaxVertexBlendMatrices < 2)
150         {
151             // No blend matrices available
152             m_use_vertex_shader = true;
153             rt::enable_menu(::GetMenu(m_hWnd),
154                 ID_OPTIONS_USECUSTOMVERTEXSHADER, false);
155         }
156         else
157         {
158             // Both techniques available
159             m_use_vertex_shader = false;
160             rt::enable_menu(::GetMenu(m_hWnd),
161                 ID_OPTIONS_USECUSTOMVERTEXSHADER, true);
162         }
163
164         compute_weights();
165
166         // if we can use vertex shaders, create the shader now
167         if ((m_dwCreateFlags & D3DCREATE_SOFTWARE_VERTEXPROCESSING)
168             || m_d3dCaps.VertexShaderVersion >= D3DVS_VERSION(1,0))
169         {
170             rt::tstring filename = rt::find_media(_T("Blend.vsh"));
171
172             // Assemble the vertex shader from the file
173             rt::dx_buffer<DWORD> code;
174     #if defined(DEBUG) || defined(_DEBUG)
175     #define RT_SHADER_DEBUG D3DXSHADER_DEBUG
176     #else
177     #define RT_SHADER_DEBUG 0
178     #endif
179             THR(::D3DXAssembleShaderFromFile(filename.c_str(),
180                 NULL, NULL, RT_SHADER_DEBUG, &code, NULL));
181     #undef RT_SHADER_DEBUG
182
183             // Create the vertex shader
184             THR(m_pd3dDevice->CreateVertexShader(code,
185                 &m_vertex_shader));
186         }
```

```
187
188     return S_OK;
189 }
190
191 ///////////////////////////////////////////////////////////
192 // RestoreDeviceObjects()
193 //
194 // Restore the mesh, cached mesh data, and other device
195 // state.
196 //
197 HRESULT CMyD3DApplication::RestoreDeviceObjects()
198 {
199     // Restore mesh's local memory objects
200     m_mesh.RestoreDeviceObjects(m_pd3dDevice);
201
202     // Get access to the mesh vertex and index buffers
203     {
204         CComPtr<ID3DXMesh> mesh = m_mesh.GetLocalMesh();
205         THR(mesh->GetVertexBuffer(&m_vertices));
206         THR(mesh->GetIndexBuffer(&m_indices));
207         m_num_vertices = mesh->GetNumVertices();
208         m_num_faces = mesh->GetNumFaces();
209     }
210
211     // Set miscellaneous render states
212     const rt::s_rs states[] =
213     {
214         D3DRS_ZENABLE, true,
215         D3DRS_AMBIENT, D3DCOLOR_XRGB(64, 64, 64),
216         D3DRS_LIGHTING, true
217     };
218     rt::set_states(m_pd3dDevice, states, NUM_OF(states));
219
220
221     // Set the projection matrix
222     const float aspect = float(m_d3dsdBackBuffer.Width) /
223         m_d3dsdBackBuffer.Height;
224     ::D3DXMatrixPerspectiveFovLH(&m_projection, D3DX_PI/4,
225         aspect, 1.0f, 10000.0f);
226     THR(m_pd3dDevice->SetTransform(D3DTS_PROJECTION,
227         &m_projection));
228
229     // look_at(eye, look at, view up)
230     m_view = rt::mat_look_at(D3DXVECTOR3(0.0f, -5.0f, -10.0f),
231                              D3DXVECTOR3(0.0f, 0.0f,  0.0f),
232                              D3DXVECTOR3(0.0f, 1.0f,  0.0f));
```

```
233        THR(m_pd3dDevice->SetTransform(D3DTS_VIEW, &m_view));
234
235        // Create a directional light
236        D3DLIGHT9 light;
237        ::D3DUtil_InitLight(light, D3DLIGHT_DIRECTIONAL,
238            -0.5f, -1.0f, 1.0f);
239        const float intensity = 0.9f;
240        light.Diffuse.r = intensity;
241        light.Diffuse.g = intensity;
242        light.Diffuse.b = 0;
243        THR(m_pd3dDevice->SetLight(0, &light));
244        THR(m_pd3dDevice->LightEnable(0, true));
245
246        // Restore the font
247        m_font.RestoreDeviceObjects();
248
249        return S_OK;
250    }
251
252    ////////////////////////////////////////////////////////////
253    // FrameMove()
254    //
255    // Change the world matrix over time; each transformation
256    // corresponds to a menu item.
257    //
258    HRESULT CMyD3DApplication::FrameMove()
259    {
260        // Update user input state
261        UpdateInput();
262
263        // Update the model rotation state
264        {
265            if (m_input.m_left && !m_input.m_right)
266            {
267                m_rot_y += m_fElapsedTime;
268            }
269            else if (m_input.m_right && !m_input.m_left)
270            {
271                m_rot_y -= m_fElapsedTime;
272            }
273            if (m_input.m_up && !m_input.m_down)
274            {
275                m_rot_x += m_fElapsedTime;
276            }
277            else if (m_input.m_down && !m_input.m_up)
278            {
```

```
279                 m_rot_x -= m_fElapsedTime;
280             }
281         }
282 
283         // Set the vertex blending matrices for this frame
284         switch (m_xform)
285         {
286         case SKIN_TR_AXIS_ROTATION:
287             {
288                 // rotate around an oscillating axis
289                 const D3DXVECTOR3 axis(
290                     2.f + std::sinf(3.1f*m_fTime),
291                     2.f + std::sinf(3.3f*m_fTime),
292                     std::sinf(3.5f*m_fTime));
293                 ::D3DXMatrixRotationAxis(&m_lower_arm,
294                     &axis, std::sinf(3*m_fTime));
295             }
296             break;
297 
298         case SKIN_TR_CENTER_SCALING:
299             {
300                 // scale about mesh center
301                 const D3DXVECTOR3 center = 0.5f*
302                     (m_mesh_bounds.m_min + m_mesh_bounds.m_max);
303                 const float s = 0.75f + 0.5f*std::sinf(3*m_fTime);
304                 m_lower_arm = rt::mat_trans(-center)*
305                     rt::mat_scale(s)*
306                     rt::mat_trans(center);
307             }
308             break;
309 
310         case SKIN_TR_Y_TRANSLATE:
311             // Y-axis translation
312             ::D3DXMatrixTranslation(&m_lower_arm,
313                     0.0f, 3*std::sinf(3*m_fTime), 0.0f);
314             break;
315 
316         case SKIN_TR_AXIS_ROTATION_LEFT:
317             {
318                 // rotate around an oscillating axis about
319                 // the mesh's minimum
320                 const D3DXVECTOR3 axis(
321                     2.f + std::sinf(3.1f*m_fTime),
322                     2.f + std::sinf(3.3f*m_fTime),
323                     std::sinf(3.5f*m_fTime));
324                 m_lower_arm =
```

```
325                 rt::mat_trans(-m_mesh_bounds.m_min)*
326                 rt::mat_rot_axis(axis, std::sinf(3*m_fTime))*
327                 rt::mat_trans(m_mesh_bounds.m_min);
328         }
329         break;
330 
331     case SKIN_TR_Y_ROTATION_LEFT:
332         {
333             // Y-axis rotation around min_x of mesh
334             m_lower_arm =
335                 rt::mat_trans(-m_mesh_bounds.m_min)*
336                 rt::mat_rot_y(D3DX_PI/4.0f +
337                     D3DX_PI*std::sinf(3*m_fTime)/8.0f)*
338                 rt::mat_trans(m_mesh_bounds.m_min);
339         }
340         break;
341 
342     case SKIN_TR_Y_ROTATION_CENTER:
343         {
344             // Y-axis rotation around center of mesh
345             const D3DXVECTOR3 center = 0.5f*
346                 (m_mesh_bounds.m_min + m_mesh_bounds.m_max);
347             m_lower_arm = rt::mat_trans(-center)*
348                 rt::mat_rot_y(D3DX_PI/4.0f +
349                     D3DX_PI*std::sinf(3*m_fTime)/8.0f)*
350                 rt::mat_trans(center);
351         }
352         break;
353     }
354     ::D3DXMatrixIdentity(&m_upper_arm);
355 
356     // post-multiply orientation transformation onto
357     // blend transformation
358     const D3DXMATRIX rot = rt::mat_rot_x(m_rot_x)*
359         rt::mat_rot_y(m_rot_y);
360     m_upper_arm *= rot;
361     m_lower_arm *= rot;
362 
363     // Set the vertex shader constants.
364     if (m_use_vertex_shader)
365     {
366         // Some basic constants
367         const D3DXVECTOR4 zero(0,0,0,0);
368         const D3DXVECTOR4 one(1,1,1,1);
369 
370         // Lighting vector (normalized) and material colors.
```

```
371         // (Use red light to show difference from non-vertex
372         // shader case.)
373         D3DXVECTOR4 light_dir(0.5f, 1.0f, -1.0f, 0.0f);
374         ::D3DXVec4Normalize(&light_dir, &light_dir);
375         const float diffuse[] = { 1.00f, 1.00f, 0.00f, 0.00f };
376         const float ambient[] = { 0.25f, 0.25f, 0.25f, 0.25f };
377
378         // compute transposed matrices used by the shader
379         const D3DXMATRIX view_proj = m_view * m_projection;
380
381         // Set the vertex shader constants; the shader uses
382         // matrices with rows/columns reversed from D3DMATRIX,
383         // so compute transpose of matrix before storing
384 #define SVSC(addr_, data_, num_) \
385 THR(m_pd3dDevice->SetVertexShaderConstantF(addr_, data_, num_))
386         SVSC(0, zero, 1);
387         SVSC(1, one, 1);
388         D3DXMATRIX transpose;
389         ::D3DXMatrixTranspose(&transpose, &m_upper_arm);
390         SVSC(4, transpose, 4);
391         ::D3DXMatrixTranspose(&transpose, &m_lower_arm);
392         SVSC(8, transpose, 4);
393         ::D3DXMatrixTranspose(&transpose, &view_proj);
394         SVSC(12, transpose, 4);
395         SVSC(20, light_dir, 1);
396         SVSC(21, diffuse, 1);
397         SVSC(22, ambient, 1);
398 #undef SVSC
399     }
400
401     return S_OK;
402 }
403
404 ////////////////////////////////////////////////////////////
405 // Render()
406 //
407 // Set either vertex shader or vertex blend state, then
408 // draw mesh.  Draw stats if requested.
409 //
410 HRESULT CMyD3DApplication::Render()
411 {
412     // Clear the viewport
413     THR(m_pd3dDevice->Clear(0L, NULL, D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,
414         D3DCOLOR_XRGB(0, 0, 255), 1.0f, 0L));
415
416     THR(m_pd3dDevice->BeginScene());
```

```
417
418         if (m_use_vertex_shader)
419         {
420             THR(m_pd3dDevice->SetFVF(s_blend_vertex::FVF));
421             THR(m_pd3dDevice->SetVertexShader(m_vertex_shader));
422             THR(m_pd3dDevice->SetStreamSource(0, m_vertices, 0,
423                 sizeof(s_blend_vertex)));
424             THR(m_pd3dDevice->SetIndices(m_indices));
425             THR(m_pd3dDevice->DrawIndexedPrimitive(
426                 D3DPT_TRIANGLELIST, 0, 0, m_num_vertices,
427                 0, m_num_faces));
428         }
429         else
430         {
431             // Enable vertex blending using API
432             THR(m_pd3dDevice->SetVertexShader(NULL));
433             THR(m_pd3dDevice->SetTransform(D3DTS_WORLD,
434                 &m_upper_arm));
435             THR(m_pd3dDevice->SetTransform(D3DTS_WORLD1,
436                 &m_lower_arm));
437             THR(m_pd3dDevice->SetRenderState(D3DRS_VERTEXBLEND,
438                 D3DVBF_1WEIGHTS));
439
440             // Display the object
441             m_mesh.Render(m_pd3dDevice);
442         }
443
444         // Output statistics
445         if (m_show_stats)
446         {
447             RenderText();
448         }
449
450         THR(m_pd3dDevice->EndScene());
451
452         return S_OK;
453     }
454
455     ////////////////////////////////////////////////////////////
456     // compute_weights
457     //
458     // Add blending weights to the mesh based on m_weight_dist.
459     // Weights are distributed along the x-axis of the model.
460     //
461     void
462     CMyD3DApplication::compute_weights()
```

```
463 {
464     // Gain acces to the mesh's vertices
465     rt::mesh_vertex_lock<s_blend_vertex>
466         lock(m_mesh.GetSysMemMesh());
467     s_blend_vertex *verts = lock.data();
468 
469     // Set the blend factors for the vertices
470     const UINT num_vertices =
471         m_mesh.GetSysMemMesh()->GetNumVertices();
472     for (UINT i = 0; i < num_vertices; i++)
473     {
474         // find fraction along x-axis extent for this vertex
475         const float a =
476             (verts[i].v.x - m_mesh_bounds.m_min.x) /
477             (m_mesh_bounds.m_max.x - m_mesh_bounds.m_min.x);
478 
479         // apply weight distribution function
480         switch (m_weight_dist)
481         {
482         case SKIN_WD_SINUSOIDAL:
483             verts[i].blend = 1.0f - std::sinf(a*D3DX_PI);
484             break;
485 
486         case SKIN_WD_TRIANGLE:
487             verts[i].blend =
488                 a <= 0.5f ? 2.0f*a : 1.0f - 2*(a-0.5f);
489             break;
490 
491         case SKIN_WD_BRACKETED:
492             verts[i].blend =
493                 (a < 0.3f) ? 0.0f : ((a > 0.7f) ? 1.0f :
494                 (a-0.3f)/0.4f);
495             break;
496 
497         case SKIN_WD_INVERTED_FRACTION:
498             verts[i].blend = 1.0f - a;
499             break;
500 
501         case SKIN_WD_FRACTION:
502             verts[i].blend = a;
503             break;
504 
505         case SKIN_WD_SIGMOIDAL:
506             verts[i].blend =
507                 1.0f/(1.0f + expf(-25.0f*(a-0.5f)));
508             break;
```

```
509            }
510        }
511    }
```

6.13 Further Reading

Transformations of Surface Normal Vectors, by Ken Turkowski, Apple Technical Report #22, July 6th, 1990.
Abstract: Given an affine 4x4 modeling transformation matrix, we derive the matrix that represents the transformation of a surfaces normal vectors. This is similar to the modeling matrix only when any scaling is isotropic. We further derive results for transformations of light direction vectors and shading computations in clipping space.