8 Mesh Rendering

Our first sample chapter will cover the fairly broad topic of mesh rendering. As we have seen in Chapter 3, there are many different ways to submit work for the GPU to perform. To introduce the beginner to a basic example of how this can be achieved, we begin with a discussion of transformation matrices and how they are used in the context of the rendering pipeline to achieve basic mesh placement within a scene. After the basics of animating an object have been covered, we extend the simple transformation matrices to allow for animation to take place within the mesh itself. A technique for vertex skinning is presented, which allows a number of transformation matrices to be used to deform the basic shape of a model for more flexibility in how a mesh can be manipulated. Finally, we discuss adding displacement mapping to our vertex skinning sample. Displacement mapping allows the higher-frequency details of a model to be stored in a texture instead of within the mesh itself, potentially allowing for improvements on the standard vertex skinning technique.

8.1 Mesh Transformations

This section will cover some of the basic rendering operations found in a typical real-time rendering application. The rendering of a triangle mesh at a desired location, orientation, and scale within a scene can be considered the absolute minimum operation that a rendering framework must be able to perform. This chapter serves as an introduction to using the Directs D 11 rendering pipeline, and also provides an example of the methodology that will be followed throughout the remainder of the book to design and implement rendering techniques.

8.1.1 Theory

In a typical 3D application, it is common for the 3D model data that will be rendered to be generated in an external modeling package for later use. This can be an artist's modeling program, a computer-aided engineering program, or any of a host of other software packages. These programs are often referred to as *digital content creation* tools, or *DCCs* for short. When the real-time rendering program is started, it loads the 3D model data from disk, processes it into an appropriate format, and then renders the model in a given scene, as needed.

To properly render the model within the application's scene, we must specify the location, orientation, and scale that we want it to appear with. Since the model is defined in object space, we need to manipulate the vertex positions prior to rasterizing the model's triangles, so that they appear where we want them to. This manipulation is executed by performing matrix multiplication on the vertex positions with specially created transformation matrices.

We will provide a brief discussion on transformation matrices here, but will not review the mathematical foundations behind their functionality. Our intention in this book is to provide the best possible resource to Direct3D 11, which means that we could not possibly provide a complete introduction to computer graphics. The formulas described below are taken from the DirectX SDK and are suitable for our uses in the current example. Since we are only performing a basic-level matrix manipulation, we can use the matrices without a complete understanding of their details. If readers are interested in further information, we refer them to (Eberly, 2007) and (Akenine-Moeller, 2002) for a more detailed discussion of transformation matrices. Unless otherwise stated, all transformation matrices are defined such that a position is represented with a 1x4 vector that is right-multiplied by a 4x4 transform matrix to produce another 1x4 row vector.

World Space

To render a triangle mesh, the first model manipulation is to determine the spatial state of the model within the scene. More specifically, we need to specify the desired scale, orientation, and position of the model. Each of these desired properties can be expressed in a 4x4 matrix form, which is commonly referred to as a *homogenous matrix representation.* As we will see later in this section, using homogenous matrices for our object manipulations allows for easier manipulations and combination of the properties they represent.

Scale matrices. The first property of the mesh that we will examine is its *scale.* This is commonly used to convert the model from its original size (as created in the artist's DCC tool) to the desired size within the rendered scene. In addition, the scale of an object can be modified to achieve some simple effects, such as shrinking, expanding, or oscillating the size of an object. The size of the model is manipulated with a scale matrix, which is created

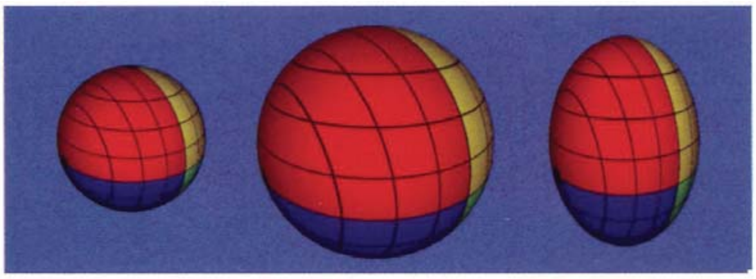
Figure 8.1. An example model (left) scaled with a uniform scaling matrix (middle) and a non-uniform scaling matrix (right).

in the form shown in Equation (8.1):

$$S = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.1}$$

In this equation, we can see that a scaling value can be applied to each of the three coordinates individually. When all three of these values are the same, the matrix is called a *uniform scaling matrix.* Uniform scaling effectively only changes the size of the object being transformed, while *non-uniform scaling* changes both its size and shape. Figure 8.1 demonstrates various scaling matrices being applied to a model, in comparison with the original model.

Rotation matrices. The next property of our model to manipulate is the orientation. Typically, the orientation of an object is modified by applying a rotation around one of the three principal axes at a time. The angles of rotation are referred to as *Euler angles,* since they were first described by Leonhard Euler. Each of these rotations can be applied with an individual rotation matrix, and the complete orientation of an object is represented by three rotation matrices—one for rotation about each of the three principal axes. Each rotation is performed about its respective axis, meaning that the overall rotation can be considered to occur around the origin of the model's frame of reference.

When dealing with rotations, care must be taken to ensure that the rotations are all applied in a consistent order, since matrix multiplication is not commutative. A rotation about each axis can be created as shown in Equation (8.2):

$$R_X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_Y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, R_Z = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{8.2}$$

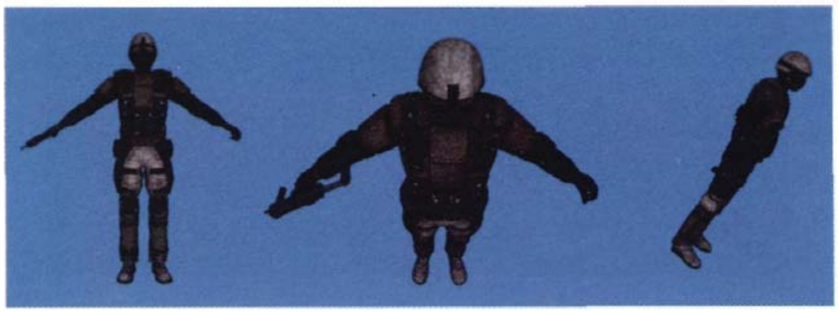
Figure 8.2. Several example model rotations are shown. Many combinations are possible, such as rotation about the *x*-axis (left), the *y*-axis (middle), or both (right). Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

In some cases, you would restrict one or more of the potential rotations, such as the camera orientation in a *first person shooter (FPS)* game that doesn't allow rotation about the Z axis. However, it is generally possible for the objects in a scene will allow rotations about all three axes. Several examples of rotations being applied to a model are shown in Figure 8.2.

Translation matrices. Finally, the position of the model within our scene can be manipulated with a *translation matrix.* This matrix is used to move an object from one location to another, without modifying the other properties discussed above. The method for creating a translation matrix is shown in Equation (8.3):

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}. \tag{8.3}$$

The translation matrix is arguably the simplest of the three operations, and it is very intuitive to consider the results of a translation operation. Several examples are provided in Figure 8.3.

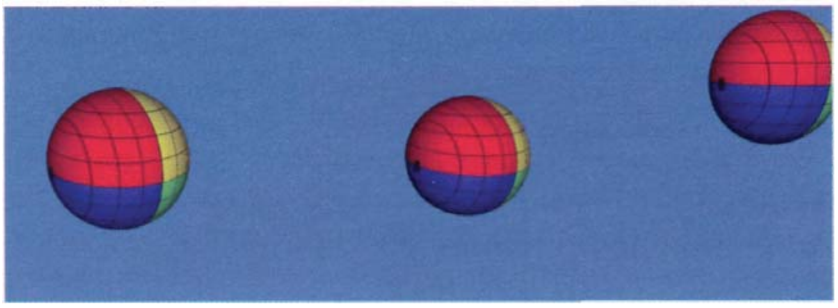
Figure 8.3. Several model translations, including movement along the x-axis (middle) and the *x*- and *y*-axes (right).

Matrix concatenation. When taken together, these three types of transformations can be concatenated into a single matrix. These transforms are applied in the order shown in Equation (8.4) when multiplying the vertex positions as a row vector on the left side of the matrices. The rotation transformation is assumed to be the concatenation of the three individual rotations, which together determine the orientation of the model. The order that these transformations are performed in is scaling, then rotation, then translation:

$$W = S * R * T. \tag{8.4}$$

By concatenating all of the matrices into a single one, it can be applied to the model in a single matrix multiplication, instead of in several individual ones. Since this combined transform converts the model from its native *object space* to the scene's *world space,* it is commonly referred to as the *world matrix.* Further consideration must be made for transforming the normal vectors of the model. Since the normal vectors are vectors and not positions, it does not necessarily produce the correct results if transformation with a scaling component is used. For a transformation matrix containing a uniform scaling, normalizing the result of the transformation is sufficient to produce the correct normal vector. However if the transformation contains a non-uniform scaling, the normal vector must be transformed with the transpose of the inverse of the transformation used for the vertex positions. This is shown in Equation (8.5):

$$N = (W^{-1})^{T}. \tag{8.5}$$

View Space

Once the model's vertices have been positioned within the scene's world space, we need to again reposition them, according to where they are being viewed from. This is typically performed by applying a *view space transformation matrix,* which represents a translation and rotation of the world space coordinates to place them into a frame of reference relative to the virtual camera that the scene is being rendered from. To construct a typical view matrix, we can use the formula shown in Equation (8.6):

$$V = T * R_z * R_y * R_x . \tag{8.6}$$

In this case, the translation actually is the negated position of the camera. This is because we are moving the world with respect to the camera instead of moving an object with respect to the world, as we did in the world space section. Likewise for the rotations, each of them represents the opposite of the rotation amount that would be applied to the camera ifit were being rendered. Direct3D 11 provides the D3DXMatrixLookAtLH() C++function in the D3DX library for constructing a view matrix based on more intuitive parameters, such as the location of the camera and the point it is looking at. Figure 8.4 shows where a

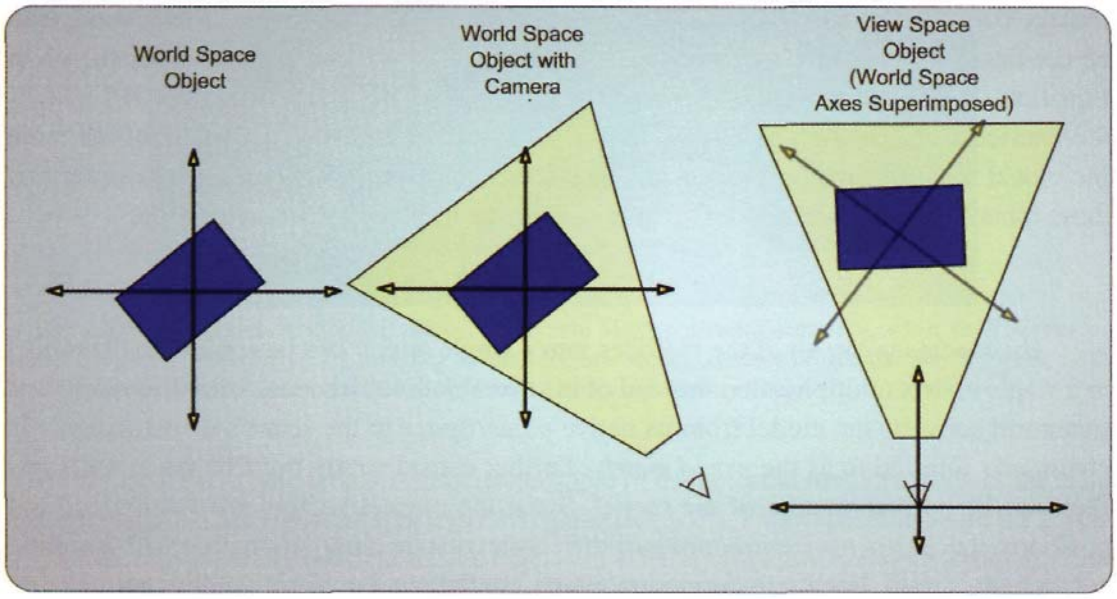
World Space Object
World Space Object with Camera
View Space Object (World Space Axes Superimposed)

Figure 8.4. A demonstration of how a view matrix is used to transform a scene into view space.

camera is located in world space, and the subsequent movement of the other objects when they are converted to view space.

Projection Space

After obtaining the model data in view space, we need to apply a projection. It is quite common in games and many visualization applications to use a perspective projection that attempts to mimic the real-world perspective. This is not the only type of projection though—many CAD packages also support orthographic projections as well. Since the focus of this book is on real-time rendering, we will focus on the perspective projection. While the view matrix defines the location and orientation of the camera viewing the scene, the projection matrix can be considered to define the camera's other characteristics, such as the *field of view (FOV),* the aspect ratio, and the near and far clipping planes to be used. These parameters define a viewing frustum, which indicates which objects are visible for a given view position and set of projection characteristics.

This projection matrix will transform our vertices into what is referred to as *clip space.* Clip space has been discussed in some detail in Chapter 3, but we can summarize the concept here. In general, clip space is the frame of reference immediately after the projection matrix has been applied. If the positions of each vertex are divided by their TV-component, they will be contained in a space that represents the viewable area of a scene within the *unit cube.* This unit cube extends from the origin of this space to +1 and -1 in

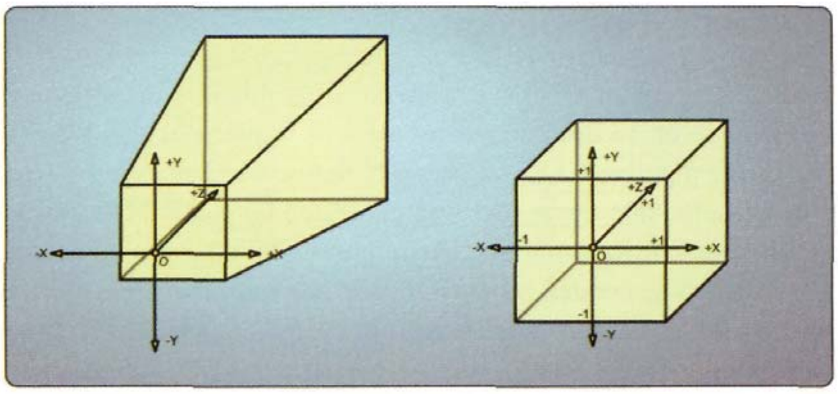

Figure 8.5. The view frustum as seen in world space, and how it maps to the unit cube.

the *x*- and *y*-directions, and from 0 to 1 in the *z*-direction. In essence, the eight corners of the viewing frustum are mapped to the eight corners of the unit cube. This means that if an object is within the viewing frustum in view space, it will end up inside the unit cube after projection and *w*-divide. Figure 8.5 demonstrates a view frustum being converted to clip space.

The projection process is accomplished through a single projection matrix. A typical projection matrix formula is shown in Equation (8.7), which is also implemented in the D3DXMatrixPerspectiveFovLH() C++ function:

$$P = \begin{bmatrix} \text{xScale} & 0 & 0 & 0 \\ 0 & \text{yScale} & 0 & 0 \\ 0 & 0 & z_f / (z_f - z_n) & 1 \\ 0 & 0 & -z_n * z_f / (z_f - z_n) & 0 \end{bmatrix} \tag{8.7}$$

$$\text{yScale} = \cot\left(\frac{\text{fovY}}{2}\right)$$

$$\text{xScale} = \text{yScale}/\text{aspect}.$$

The three different transformation matrices we have discussed up to this point—the world, view, and projection matrices—can also be concatenated into a single complete transform matrix. When they are used to render a large number of vertices (as we will be doing), applying a single matrix instead of individually multiplying each matrix on its own will provide a significant performance benefit by reducing the number of calculations performed, while still producing the same result.

8.1.2 Implementation Design

With a clear understanding of what we want to accomplish with our transformation matrices, we can move on to designing how we will implement this algorithm in the Direct3D 11 rendering pipeline. The first step is to define what our input model data will look like. We can assume that the model was generated by some DCC tool in a format that can be read by our sample program. The model will be a static model, with vertex positions, normal vectors generated by the tool, and a texture map applied to the model. This implies that we will have per-vertex texture coordinates as well. The specific vertex format is shown in Figure 8.6.

Now that we know what information we will be storing, we must make a decision about how to structure the resources that will hold our model data. These resources will be bound to the pipeline as inputs to the input assembler stage. It is common to store our model data in one buffer resource to hold the per-vertex data (referred to as a *vertex buffer),* and one buffer resource to hold our primitive specification data (referred to as an *index buffer).* Let's consider the vertex buffer first. As shown in Figure 8.6, in this sample application we will use a per-vertex position, normal vector, and texture coordinate for our vertex format. The vertex buffer will simply contain an array of these structures to represent each of the vertices of the input model.

Next, we will look at our index buffer. There are many primitive types available for use in Direct3D 11, especially when you add in the control patch formats used with the tessellation system. There are also different index ordering semantics involved with each of the primitive types. For example, a triangle list uses three indices for each triangle primitive, while a triangle strip only uses two indices plus one additional index per primitive. For our first implementation, we will start with a basic triangle list primitive type, for ease in specifying and inspecting data. Thus, the index buffer will provide a list of indices, which point into the vertex buffer. The total number of indices is indicated by multiplying the number of triangles by three.

With the input buffers defined, we can consider how the output of the pipeline will be configured. The normal output configuration is to have both a render target and a depth stencil buffer bound to the pipeline in the output merger stage. The render target is typically acquired from the swap buffer that is being used to present the rendered output to a window. Unless otherwise noted, this is the configuration that will be used in the subsequent examples.

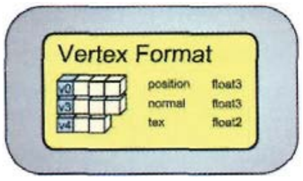

Figure 8.6. A visualization of the vertex format that we will be using.

If we consider our current pipeline design, we can see that the input resources and output resources have been identified. Now we need to make a decision regarding which portions of the pipeline we will use to implement this algorithm. As noted before, we will be applying our transformation

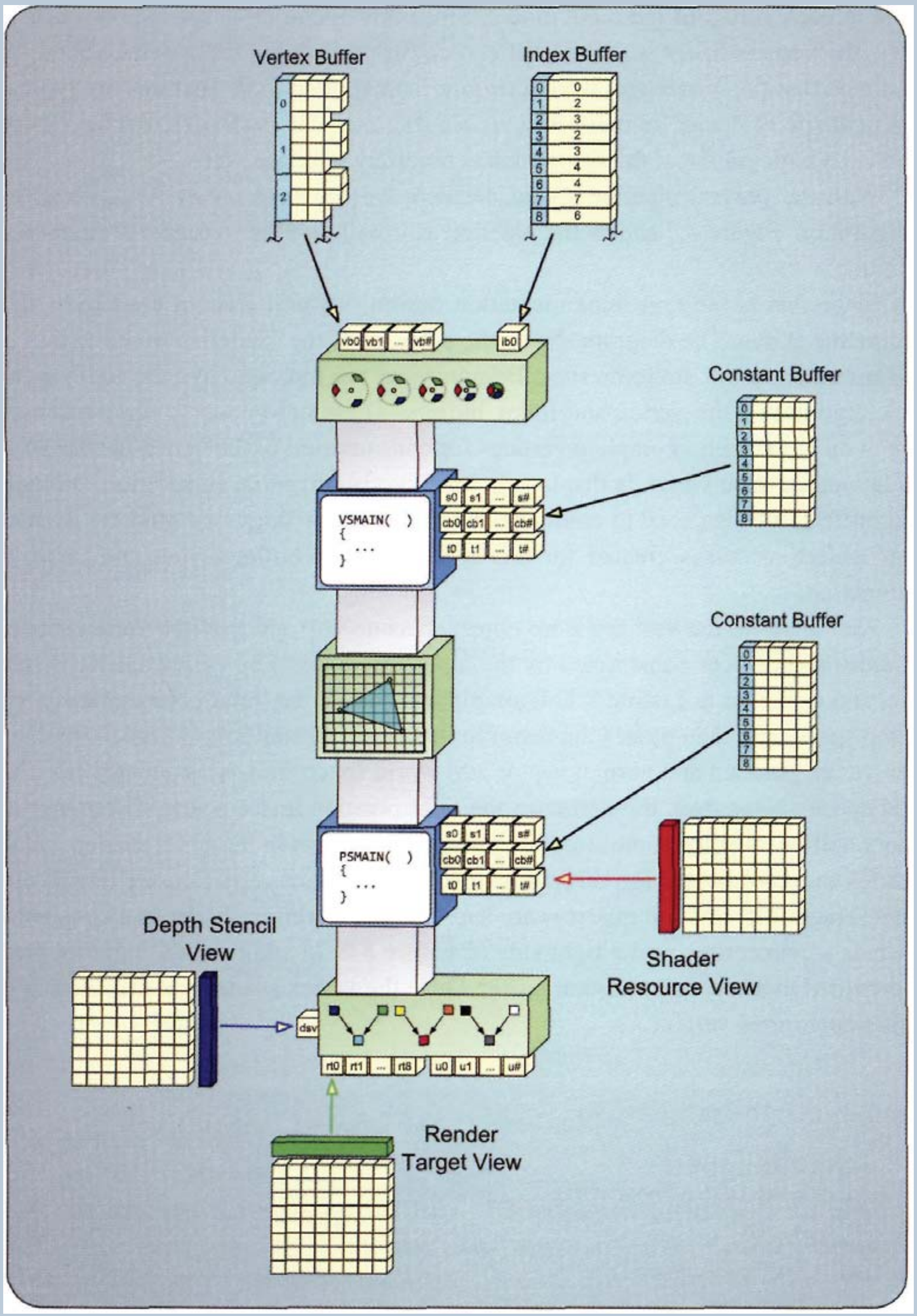

Figure 8.7. The pipeline configuration for rendering a static triangle mesh in our current rendering scheme.

matrix to each vertex of the input model. Since this operation is carried out once on each vertex, the vertex shader is the natural choice for performing these calculations. We also mentioned that the model will have a texture map applied to it. Textures are typically applied in the pixel shader, so that every visible fragment can perform a texture lookup. This results in a high graphical fidelity, which is precisely what we want.

With the general pipeline layout decided, we can now review the overall pipeline configuration. Figure 8.7 shows the pipeline as it will be used to render a model with this technique.

Since this is the first implementation design, we will discuss each state shown in the pipeline above. The diagram shows the portions of the rendering pipeline that we will use for rendering our static meshes. Beginning at the top, we have the input data to the whole algorithm—the vertex and index buffers. These are bound to the input assembler stage, which assembles complete vertices for consumption in the vertex shader stage. The data layout for each vertex is displayed, for easy visualization. In addition to binding the input buffers, we also need to configure the primitive topology and bind the current input layout object, which is created for this specific vertex buffer layout and vertex shader combination.

We transform the vertices from object space to clip space in the vertex shader, so it consumes the vertices constructed by the input assembler. The vertex shader is relatively basic, and is shown in Listing 8.1. It simply transforms the input object space position to the clip space, and then passes the result to its output. In addition, it transforms the object space vertex position and normal vector into world space, and subsequently calculates the world space vector from the vertex to the light position in the scene. These world space vectors will be used for some simple lighting calculations in the pixel shader. Finally, the input texture coordinates are directly copied to the output structure for use in the pixel shader. The transformation matrices are supplied to the vertex shader in a constant buffer, shown as a connection on the right side of Figure 8.7. In addition, the lighting properties are provided in a separate constant buffer. Once the vertex shader has executed, it emits a single transformed vertex.

```
cbuffer StaticMeshTransforms
{
    matrix WorldMatrix;
    matrix WorldViewProjMatrix;
};

cbuffer LightParameters
{
    float3 LightPositionWS;
    float4 LightColor;
};
```

```
Texture2D       ColorTexture : register( t0 );
SamplerState    LinearSampler : register( s0 );


struct VS_INPUT
{
    float3 position  : POSITION;
    float2 tex       : TEXCOORDS0;
    float3 normal    : NORMAL;
};

struct VS_OUTPUT
{
    float4 position  : SV_Position;
    float2 tex       : TEXCOORD0;
    float3 normal    : NORMAL;
    float3 light     : LIGHT;
};

VS_OUTPUT VSMAIN( in VS_INPUT input )
{
        VS_OUTPUT output;

        // Generate the clip space position for feeding to the rasterizer
        output.position = mul( float4( input.position, 1.0f ), WorldViewProjMatrix );

        // Generate the world space normal vector
        output.normal = mul( input.normal, (float3x3)WorldMatrix );

        // Find the world space position of the vertex
        float3 PositionWS = mul( float4( input.position, 1.0f ), WorldMatrix ).xyz;

        // Calculate the world space light vector
        output.light = LightPositionWS - PositionWS;

        // Pass through the texture coordinates
        output.tex = input.tex;

        return output;
}
```

Listing 8.1. The vertex shader for static mesh rendering.

Since no hardware tessellation is needed in this technique, we can skip the hull, tessellator, and domain shader stages. We also don't need per-primitive level manipulations of the model data, so the geometry shader stage is also not used. This means that the vertices emitted from the vertex shader are directly consumed by the rasterizer stage. Once the rasterizer receives a primitive (consisting of three vertices for our triangle lists) it executes, and produces fragments to be consumed by the pixel shader. Each fragment uses a format similar to that of

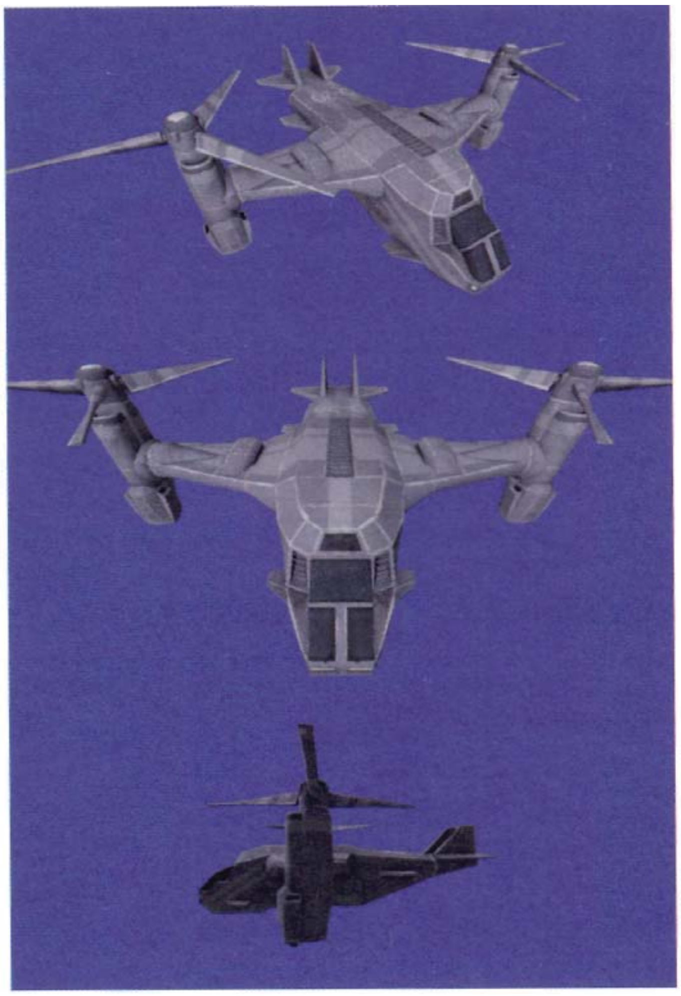

Figure 8.8. The final results of performing the rendering of a static mesh. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

the vertices consumed by the rasterizer, except that the position information is no longer needed and hence doesn't appear in the output format of the rasterizer.

The pixel shader receives these fragments from the rasterizer and must determine the color of the model at each fragment. The pixel shader is shown in Listing 8.2. We want the pixel shader to sample a texture map to determine the surface properties of the mesh. As described in Chapter 2, the Texture2D resource requires a shader resource view to bind it to the pixel shader stage for read-only access. The pixel shader samples the texture and then calculates the amount of illumination that is available at this point on the surface with the dot product of the normal and light vectors. Finally, the surface color is modulated by the illumination value, and the result is emitted as output color information to the output merger stage. Finally, the output merger writes the output color data to the attached render target, as well as the depth stencil resource, after performing the depth test (the stencil test and blending functionality are not used).

```
float4 PSMAIN( in VS_OUTPUT input ) : SV_Target
{
      // Normalize the world space normal and light vectors
      float3 n = normalize( input.normal );
      float3 l = normalize( input.light );
      // Calculate the amount of light reaching this fragment
      float4 Illumination = max(dot(n,l),0) + 0.2f;

      // Determine the color properties of the surface from a texture
      float4 SurfaceColor = ColorTexture.Sample( LinearSampler, input.tex );

      // Return the surface color modulated by the illumination
      return( SurfaceColor * Illumination );
}
```

Listing 8.2. The pixel shader for static mesh rendering.

Figure 8.8 shows the results of our rendering as we have configured it for several orientations of the model and camera.

8.1.3 Conclusion

Rendering a static mesh is somewhat of a first point of entry into developing real-time rendering algorithms. This technique allows for an object to become animated by changing some of its properties over time. For example, if the object were spinning in a circle, this could easily be accomplished by animating the rotation about the *y*-axis. The operation described above would be executed once for each object that needs to be rendered in a given scene. Since we are using a transformation matrix and a texture map that will be unique for each object, each object must be rendered with its own draw calls after the associated resources have been updated and bound to the pipeline.

8.2 Vertex Skinning

In the previous section, we saw how to perform the transformation of an object to control its spatial properties within a scene. This provides a surprisingly large amount of functionality. For instance, it could be used to implement a chess game where each piece is represented by a static mesh that only moves within the scene. However, many other types of objects would require a different type of animation to appear correct. For example, an object composed of multiple pieces that move with respect to one another (like a robotic arm) can't be accurately portrayed with a static mesh. We could use multiple individual meshes and manipulate their transform properties to provide a more convincing result, but this would not be applicable to any type of organic mesh, such as an animal or a humanoid figure. Static meshes with transformations are simply not capable of properly rendering this class of objects.

Instead, another technique must be used: *vertex skinning*. This algorithm lets individual portions of a triangle mesh be animated with their own object space, prior to the transformation matrix being applied. This allows each of the scenarios described above to be more convincingly rendered, and provides a flexible method for incorporating animation data directly into the model, instead of having to manually control every movement of the object at a higher level. In this section, we will introduce the concept of vertex skinning, and the theory behind its operation. We will develop an implementation of this algorithm for Direct3D 11, and will finally consider the performance implications of such an approach.

8.2.1 Theory

The key concept behind vertex skinning is the ability to use transformation hierarchies. In the previous section, we learned how an object can be placed throughout a scene in various orientations or scales. This rendering technique allows each object to be positioned with respect to the world space origin. However, if we consider the robotic arm example from above, if there are multiple meshes that need to be positioned relative to another object in the scene instead of the origin, it can quickly become difficult to manage two different world-space transforms that depend on one another. This would be even more difficult if there are more than two joints in the arm. Figure 8.9 shows such an arm, where the components of the arm that are connected together can only move with respect to the joint that they are attached to.

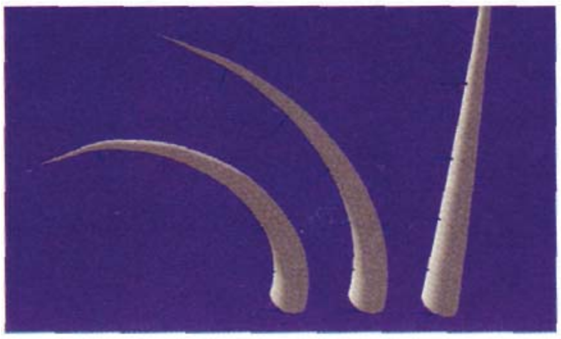

Figure 8.9. An arm that demonstrates the concept of multiple mesh sections constrained to one another.

Transformation Hierarchies

The type of movement constraint that we need can be accomplished with *transformation hierarchies.* If one object's transformation matrix is generated such that it is applying the position and orientation of its mesh with respect to the mesh that it is attached to, it can be rendered by concatenating its own world space matrix with that of its "parent" object. Returning to our robotic arm example, if each of these arm components is connected to a parent object, it only needs to manage the transformation state with respect to its parent. When it is time to render each component, we would begin with the component that has no parent and calculate its transformation matrix. This would be followed by the first child calculating its transformation matrix and then concatenating this value with the parent's transformation matrix. This process of concatenating matrices essentially moves the child object from a coordinate space defined by its parent to the world space in which its parent is located. This is repeated until all of the components have been updated, at which point we could render them individually with their newly calculated world matrices.

This concept was used in computer graphics long before programmable GPUs were developed. However, this is still an interesting topic to consider, which can be applied to the more difficult problem of rendering organic objects. Using transformation hierarchies, we can render individual portions of the robotic arm, since they actually are individual pieces. However, a human arm not only has bones that are separate pieces, but also has a muscular system and skin that are continuous across the arm. This doesn't lend itself to rendering an arm in pieces, since the object should appear continuous, as it does in the real world.

Transformation hierarchies still hold the key to finding a solution to this problem. Instead of trying to split the arm into multiple discrete pieces, it would be better to be able to say how much each vertex is influenced by each the objects it is in contact with. For example, the skin on an elbow moves partially with the upper arm and partially with the lower arm. If we develop a system that allows us to assign a weighting for each vertex to each specify how much it is influenced by a particular component, we could calculate the vertex position as if it were attached to each component and then interpolate its final position based on the weights applied to each component.

Skin and Bones

This is the primary idea behind vertex skinning. The vertices of a mesh define the *skin* of a model, and each component included in the transformation hierarchy is referred to as a *bone.* The complete group of bones is referred to as the *skeleton* of the model. These names provide a simple visualization of what the individual pieces of this algorithm represent. There are many different variations of this algorithm that can be more or less suitable for a given situation, but the algorithm provided here is a fairly general version. We will allow each vertex to be associated with up to four bones each, which will allow for a significant amount of flexibility in making an animation look correct.

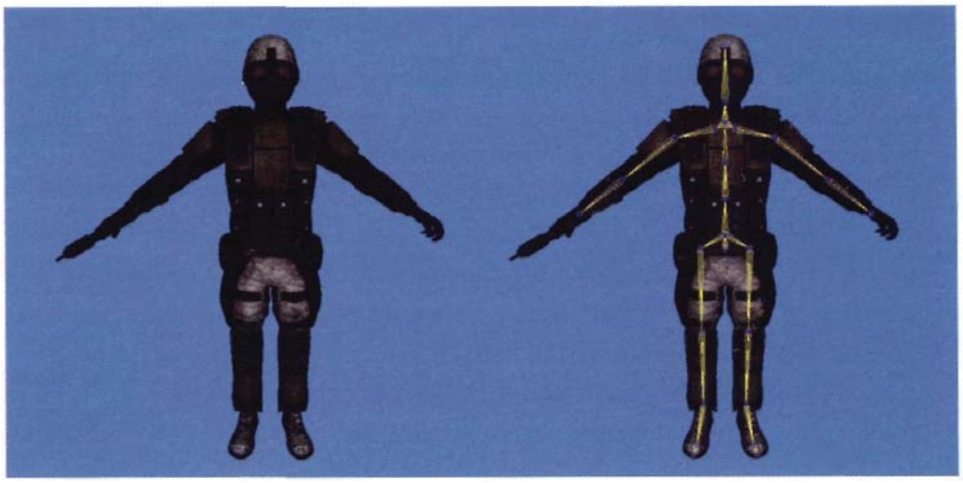
Figure 8.10. A sample mesh shown with and without its skeleton. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

To develop a model for rendering, we need to determine the number of bones that a model will use, and how they will be connected. This is typically determined in the design stage and can be chosen to suit the needs of a particular application. It is quite common for humanoid models to define their bones in similar locations to where the human body has joints between bones. The root of the bone hierarchy is typically chosen somewhere in the lower back, where it can be used to position the entire object within world space. The current bone transformation for each bone is the calculated by multiplying the transformation matrix from the root bone with each successive bone's transformation matrix, until you reach the current joint. An example mesh is shown with and without its skeleton in Figure 8.10.

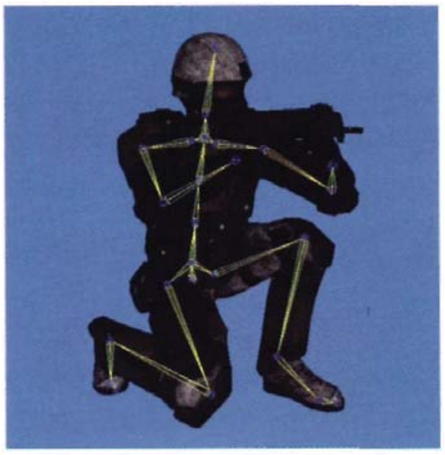
Figure 8.11. A sample model shown in a crouching pose. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

In this case, we see the skeleton oriented in what is referred to as its *bind pose*. This is the base orientation of all of the bones that a model will be created in, and all of the vertices will have their individual bones selected, and weights applied, in this base orientation. After the model has been created in this manner, the individual animations can be created using the base information. Instead of trying to manually place each vertex, the animator can simply manipulate the position and orientation of the bones to create the desired pose of the model. As an example, our sample model can be seen in a crouching pose in Figure 8.11.

Mathematics of Skinning

Bind poses are constructed primarily to allow the content creator to have a simple, easy-to-understand method of creating the model. However, the use of bind poses has some implications for the mathematics of our solution. When the vertices are created and placed around a model, the positions are defined relative to the coordinate space of the entire object—not with respect to its bone. This means that before we can apply the hierarchical bone transformation matrix to a vertex, it must be converted from an object space position into a *bone space position.* However, since there are multiple bones allowed per vertex, this can't be done in the modeling package. Instead, this must be performed during runtime, when each bone transformation matrix is created.

Fortunately, matrix transformations can easily perform this kind of operation. Essentially, we are trying to "undo" the bone transformations of the bind pose, and then reapply the bone transformations of the currently desired pose. This is done by creating a transformation matrix that applies the inverse of the bind pose transformation before applying the desired pose transformation matrices. This is shown in Equation (8.8), where the final transformation matrices for each bone are the product of the inverse of the bind pose and the current bone transformation matrix:

$$\boldsymbol{B}[\boldsymbol{n}]_{\text{final}} = \boldsymbol{B}[\boldsymbol{n}]^{-1}_{\text{bind}} * \boldsymbol{B}[\boldsymbol{n}]_{\text{curr}}. \tag{8.8}$$

We can visualize what this series of transforms does by considering the vertices associated with a hand bone. In the bind pose, the vertex positions are specified in the object space of the model. When the inverse of the bind pose is applied to the vertices, the hand bones moves to the origin of the coordinate space, so that it would be sticking out of the origin at the wrist. Then the current bone pose transformation moves the hand bone to the desired location. Since the bind pose is always the same throughout the lifetime of an application, it can be calculated once at startup and then simply applied to the hierarchical bone transformation matrices that are calculated for each frame of animation.

8.2.2 Implementation Design

With a clear concept in mind about how vertex skinning works, we can now design an implementation to be used with Direct3D 11. We can use a similar rendering configuration to what we saw in the static mesh rendering section, with two exceptions: we need to incorporate the new per-vertex data into the vertex structure, and then the portion of the vertex shader that transforms the vertices must be updated to use an array of bone matrices and perform the bone-weighting interpolation. We will take a closer look at each of these steps. The updated algorithm layout is shown in Figure 8.12 for reference during the discussion.

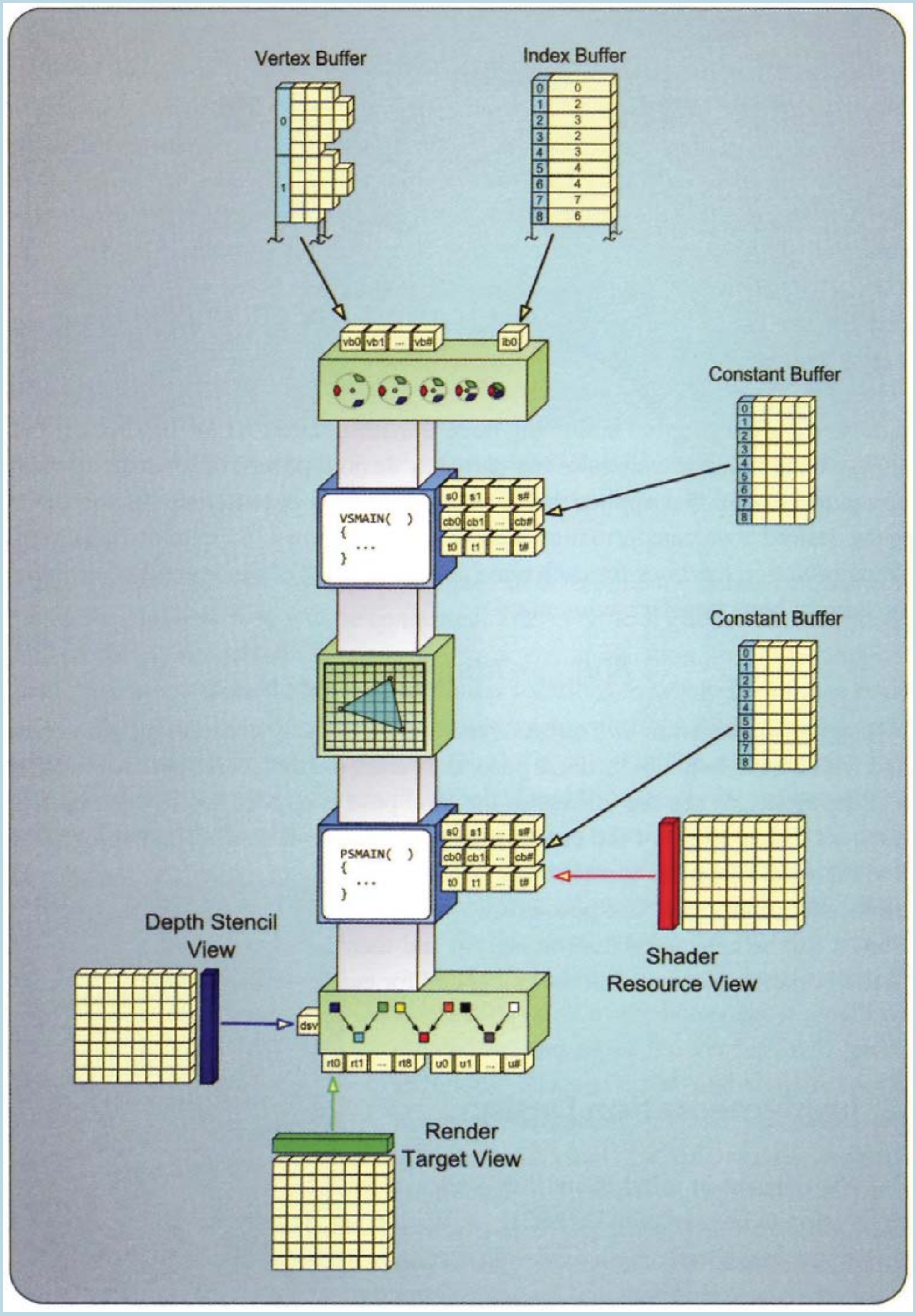

Figure 8.12. The vertex-skinning pipeline configuration.

To support the vertex skinning algorithm, our model importer must support a different type of per-vertex information. The new vertex format must include the information that specifies which bone it is associated with, as well as what weighting amount each bone should have on the outcome of the interpolation. To accomplish this, each bone must be assigned an integer index that can be used as an index into an array of matrices. We will allow up to four bones to influence each vertex, which indicates that we will need a four-component set of bone indices, as well as a four-component set of bone weights. The updated vertex structure is shown in Figure 8.13. Of course, with a new vertex structure a new input layout object must be generated as well.

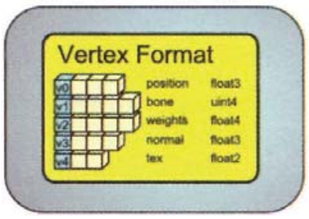

Figure 8.13. The vertex layout for our vertex-skinning implementation.

To use the new vertex data, we must modify our vertex shader according to the algorithm we specified in the theory section. The first step is to gain access to the transformation matrices that the application provides. The matrices are made available to the vertex shader through a constant buffer, and essentially replace the world transformation matrix portion of the concatenated transform from our previous example. This means that the will calculate the world-space position of each vertex within the vertex shader, and then transform this resulting position by the view and projection matrices that will be concatenated into a single matrix. In addition to the bone transformation matrices, we also include the normal vector bone transformation matrices, to produce a correct world-space skinned vertex-normal vector as well. The modified vertex shader is shown in Listing 8.3.

```
cbuffer SkinningTransforms
{
    matrix WorldMatrix;
    matrix ViewProjMatrix;
    matrix SkinMatrices[6];
    matrix SkinNormalMatrices[6];
};

cbuffer LightParameters
{
    float3  LightPositionWS;
    float4  LightColor;
};

Texture2D       ColorTexture  :  register( t0 );
SamplerState    LinearSampler  :  register( s0 );

struct VS_INPUT
{
    float3  position   : POSITION;
    int4    bone       : BONEIDS;
    float4  weights    : BONEWEIGHTS;
    float3  normal     : NORMAL;
    float2  tex        : TEXCOORDS;
};
```

```
struct VS_OUTPUT
{
    float4 position    : SV_Position;
    float3 normal      : NORMAL;
    float3 light       : LIGHT;
    float2 tex         : TEXCOORDS;
};

VS_OUTPUT VSMAIN( in VS_INPUT input )
{
    VS_OUTPUT output;

    // Calculate the output position of the vertex
    output.position  = (mul( float4( input.position, 1.0f ),
      SkinMatrices[input.bone.x] )
                          * input.weights.x);
    output.position +=(mul( float4( input.position, 1.0f ),
      SkinMatrices[input.bone.y] )
                          * input.weights.y);
    output.position += (mul( float4( input.position, 1.0f ),
      SkinMatrices[input.bone.z] )
                          * input.weights.z);
    output.position += (mul( float4( input.position, 1.0f ),
      SkinMatrices[input.bone.w] )
                          * input.weights.w);

    // Transform world position with viewprojection matrix
    output.position = mul( output.position, ViewProjMatrix );

    // Calculate the world space normal vector
    output.normal  = (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.x] )
                        * input.weights.x).xyz;
    output.normal += (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.y] )
                        * input.weights.y).xyz;
    output.normal += (mul( input.normal, (fioat3x3)SkinNormalMatrices[input,
      bone.z] )
                        * input.weights.z).xyz;
    output.normal += (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.w] )
                        * input.weights.w).xyz;

    // Calculate the world space light vector
    output.light = LightPositionWS - output.position.xyz;

    // Pass the texture coordinates through
    output.tex = input.tex;

    return output;
}
```

Listing 8.3. The vertex shader used in the vertex skinning operation.

As we can see in Listing 8.3, the input position is calculated by transforming the object space position for each of the bones specified in the vertex's bone ID attribute, and the resulting position is scaled by the bone weighting value from the vertex's bone weight attribute. The normal vector is transformed in a similar manner, except with the inverse transpose of the transformation matrix. Finally, the four weighted results are added together to find the final world space position and normal vector. The world space position can then be transformed into clip space with the ViewProjMatrix parameter, and the remainder of the vertex shader is the same as in the static mesh rendering case.

This implementation provides a few additional indicators of how the various parts of the rendering pipeline are typically applied to particular aspects of a model's appearance. For example, we modified how the geometry of a model is calculated, which required only a change to the vertex shader. The remainder of the pipeline configuration (other than the input layout) is the same. This is often the case when the geometric properties of a model are determined in the earlier stages and the surface properties of the geometry are determined in the later stages. With this separation of duties, we can use individual shader programs for more than one pipeline configuration, when it is appropriate. This reduces the number of different shader programs that need to be written. It is a good idea to keep this in mind when developing a new algorithm, since a shader program that is already written is much faster to develop than one written from scratch!

8.2.3 Conclusion

If we consider how vertex skinning operates, we can gain some insight into how the performance of the algorithm will vary with different input models. If we take the static mesh algorithm as our baseline for performance, then add vertex skinning, we must upload a larger constant buffer (to hold the bone matrices), and must perform some additional math on the vertex before it is passed out of the vertex shader. Since we are allowing four bones per-vertex, our vertex shader is written to access and calculate one transformed position for each bone, which is then weighted and summed to find the final location. Even if fewer than four bones are used in each vertex (if some of the weights are equal to 0), they must all still be calculated, regardless of if the resulting weighted position produces an all-zero position that will not contribute to the output position. Therefore, the cost associated with performing vertex skinning should be constant from vertex to vertex.

This also means that the algorithm's performance will scale with the number of vertices included in the model. If a highly detailed geometry model is rendered, its corresponding rendering cost will be proportionally higher than that of a less-detailed geometric model. Depending on the overall pipeline configuration and the hardware that an application is running on, this can limit the available model detail that can be used for a given scene. In many cases this may not make any difference, but if geometry processing is the bottleneck of the algorithm, vertex skinning may introduce a performance limitation. Luckily, there are techniques available that can reduce this dependency on vertex counts.

8.3 Vertex Skinning with Displacement Mapping

By using vertex skinning, we have significantly expanded the number of objects we can represent with our rendering algorithms. We have techniques for both static and dynamic meshes, and can perform animation with both of these mesh types. However, as mentioned in the last section, using vertex skinning can introduce a performance bottleneck, due to the additional mathematic operations performed on each vertex. It would be better to diminish the link between geometric complexity and the number of vertices that must be processed. Fortunately, some of the new abilities in the Direct3D 11 rendering pipeline can help us achieve this decoupling.

Ideally, we would like to perform the skinning operation on a low-resolution model, but we still want the detailed look of a high-resolution model. The tessellation stages of the rendering pipeline allow us to perform this geometry detail injection. There are many available techniques, but we will focus on one of the more versatile solutions—*displacement mapping*. Displacement mapping essentially applies a height-map to every triangle in a triangle mesh. By using the tessellation stages, we can tessellate a low-resolution mesh to generate a higher number of vertices, and then sample the displacement map and modify the position of each of the vertex with respect to the original lower-resolution triangles.

This effectively moves the highly detailed geometry out of the static vertex information and into a texture instead. This has many advantages. The first is that we can perform skinning on the lower-resolution geometry, but there are other benefits. Since a displacement map can be mip-mapped, it allows for a simple level of detail (LOD) system to be implemented automatically. In the following section, we will see how this technique works, and how to implement it in the context of the Direct3D 11 rendering pipeline.

8.3.1 Theory

To displace the surface of the low-resolution triangle mesh, we must perform two operations. The first is to determine how much detail to add into the model, and the second is to understand more precisely how we can perform this displacement operation. We will consider these two items before moving on to the implementation details of this algorithm.

Dynamic Tessellation

Since we will be dynamically determining how much detail to add into the lower-resolution mesh, we must consider the appropriate measures to select how much geometry is

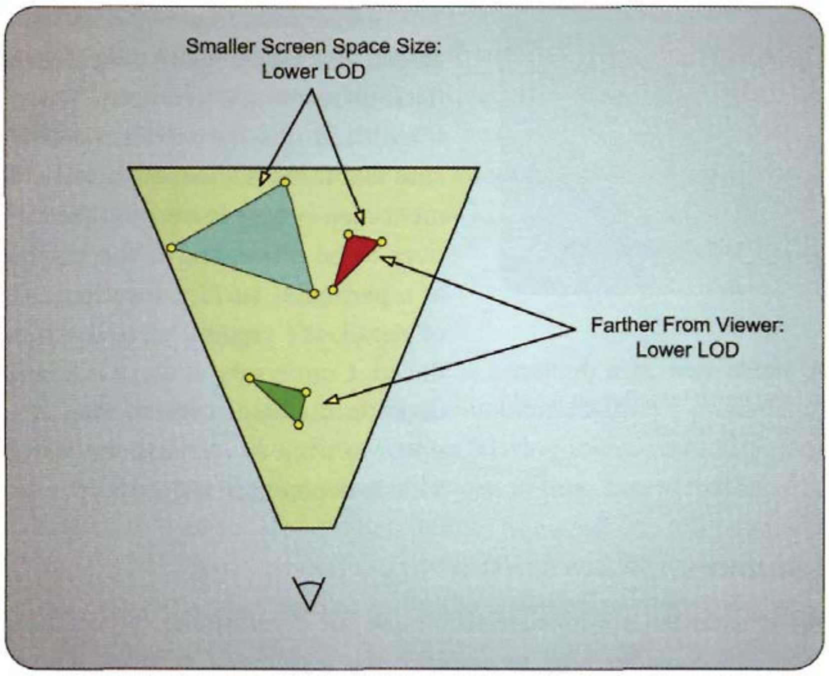

Figure 8.14. A number of different triangle properties used for determining the appropriate tessellation level.

needed for a given triangle. There are many metrics that we could use, so we will consider what situations would require the highest level of detail.

Clearly, when a triangle face points away from the viewer, we don't want to increase its complexity at all. In fact, if possible, we want to eliminate it from further processing completely, since it won't contribute to the rendered output image. At the transition point from a back-facing triangle to a front-facing triangle, we have silhouette edges. These edges have very high importance, because the user will be able to fairly clearly differentiate the outline of the object from its background. If there is a low-resolution silhouette, the user will spot it quite easily. After the silhouette edge, we have the triangles that are potentially visible to the user as long as they are not occluded by other geometry within the scene. If a triangle is potentially visible, we want to increase its level of detail to the highest degree that will be visible under the current viewing conditions. This particular type of situation provides a number of potential metrics.

The simplest property is the distance from the camera. If a triangle is very close to the camera, it should use the highest level of detail possible. If it is very far away from the camera, it should use a much lower detail. This calculation depends on the average size of a triangle, as well as the angle the triangle is being viewed with. When combined, these parameters describe the effective screen-space size of the triangle. Figure 8.14 demonstrates the various situations we have described up to this point.

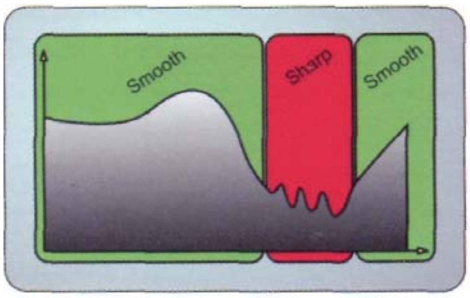

Figure 8.15. A profile view of a displacement map, and how much tessellation would be appropriate for each region.

In addition to a distance-based metric or a screen-space-size metric, we must also consider the high resolution geometry. When geometry is introduced into the model, we also want to ensure that there is adequate detail in the displacement map before increasing the number of vertices to be processed. If the displacement map at a particular surface location has a low level of detail, the amount of tessellation can be reduced. Conversely, if there is a large amount of detail in the displacement map at a given location, we want to increase the tessellation level. This is depicted in Figure 8.15.

Displacing Surfaces

Once we have chosen the appropriate technique for determining the required amount of tessellation, we can consider how to displace the tessellated vertices. The displacement function actually doesn't need to know the method that was used to determine the tessellation level; it is only concerned with taking a vertex location and finding the required displacement to apply. This operation is slightly more complicated than it may initially seem, because we are applying a flat texture to a semi-flat triangle mesh. Since neighboring triangle faces are typically not coplanar, there is at least some change in orientation between them. This makes it impossible to perform a simple displacement away from the surface due to either a gap or an overlap at the transition between faces. This is shown in Figure 8.16.

In some respects, this is why normal vectors are specified for each vertex, instead of for each triangle. The triangle mesh is an approximation of a smooth surface. Thus, the vertex normal vectors are used to make surface lighting appear smoother than it really is by

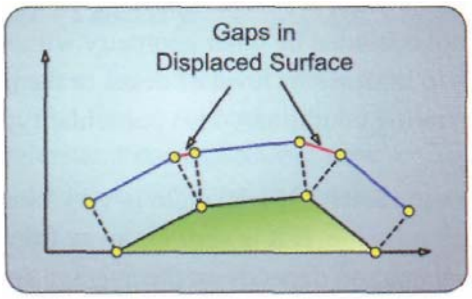

Figure 8.16. A profile of a triangle mesh, showing that non-coplanar triangles can't use simple surface offsets.

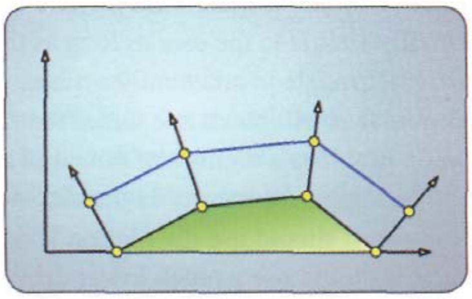
Figure 8.17. Using vertex normal vectors for displacing tessellation-generated vertices.

providing a vector that is a combination of the triangle normal vectors that it is touching. Displacement mapping can follow the same methodology by using the vertex normal vector. For each point on a triangle mesh being displaced, we read the displacement amount from the displacement map and then shift the new vertex along its interpolated vertex normal vector. This allows the displacement to be smoothly transitioned between triangle faces. This is updated scheme shown in Figure 8.17.

8.3.2 Implementation Design

Once again, we will be able to use our previous implementation's pipeline configuration, with just a few changes. Even though we will be passing lower-resolution model data into the pipeline, the vertex skinning process is oblivious to this, except that the final projection to clip space must be moved to later in the pipeline. Since we are still using skinning in the same general way on our input vertices, the vertex layout and the vertex shader will remain mostly the same. However, since we are using the tessellation stages, we must change our input topology setting to indicate that we are using control patches instead of triangles. In effect, there is no difference for our input buffers, since we will be using a three-point control patch list, which is topologically the same as a triangle. However, the runtime will require a patch list type as input to the hull shader, so we will make the change. Since we will be using the tessellation stages to increase the resolution of our geometry, the hull shader, tessellator, and domain shader all must be used, since they are used in conjunction. Finally, the pixel shader will remain the same, to apply the color texture to the geometry that is produced in the tessellation stages. The updated pipeline configuration is shown in Figure 8.18.

With the vertex shader remaining the same except for the removal of the clip space projection, the first new step to implementing our displacement-mapping addition is to configure the hull shader to set up the tessellation process. The hull shader will receive skinned vertices from the vertex shader and will interpret them as three point control patches. The hull shader requires two different functions—its main function, as well as a patch constant function. Since the patch constant function is responsible for determining the tessellation factors, we don't need to do anything in the hull shader. This lets us simply pass the hull shader input control points as the output control points, which will also remain as three-point control patches.

We will use the patch constant function to determine how finely we should split up the input control patches, by specifying one edge-tessellation factor for each edge of the triangle patch, plus a single tessellation factor for the interior portion of the triangle. Each of the metrics discussed in our theory section could be used to determine the necessary tessellation level, but we will restrict our patch constant function to using a fixed value for the amount of tessellation. This fixed value is passed to the output of the patch constant

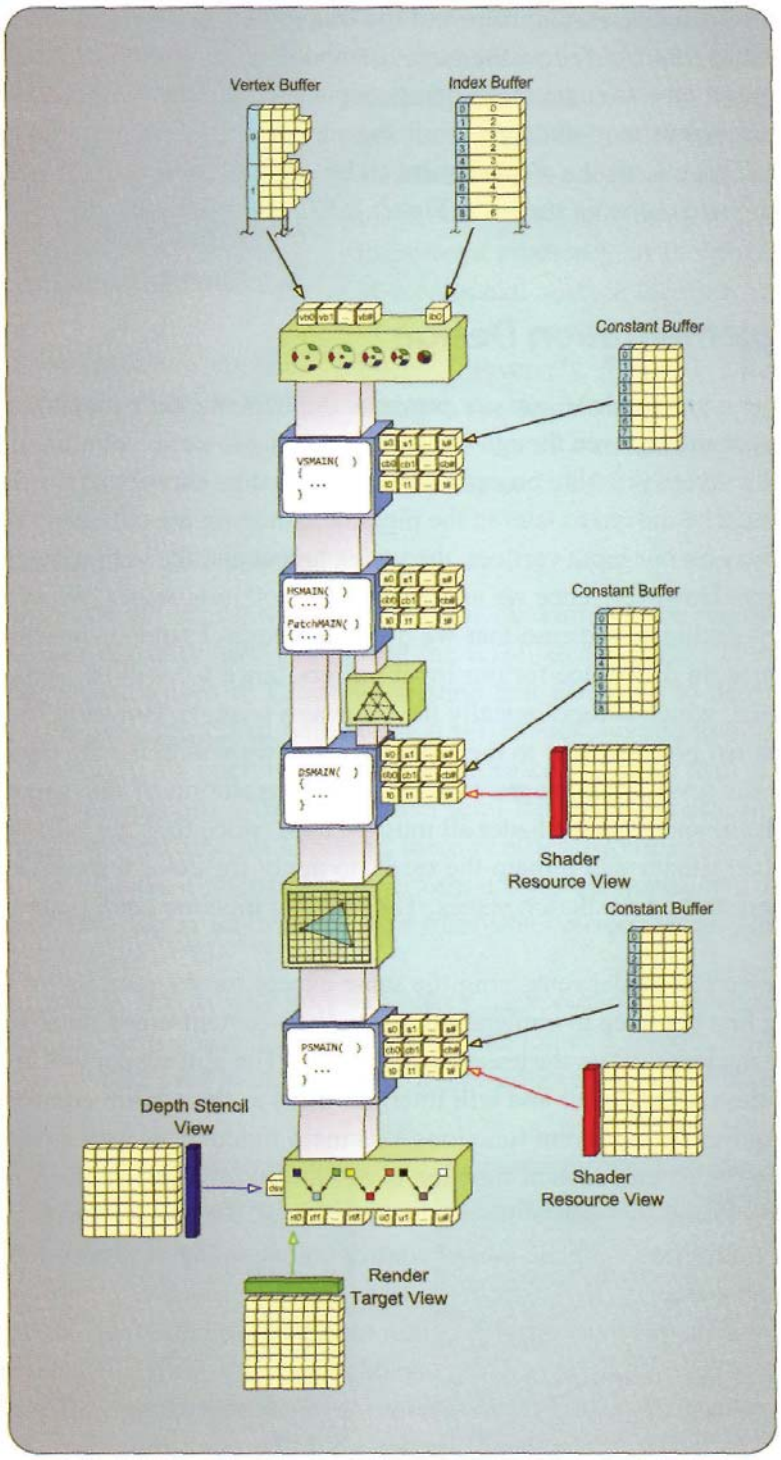

Figure 8.18. The pipeline configuration for vertex skinning with displacement mapping.

function, including each of the tessellation factor system value semantics. Both of the functions for the hull shader are shown in Listing 8.4.

```
cbuffer SkinningTransforms
{
    matrix WorldMatrix;
    matrix ViewProjMatrix;
    matrix SkinMatrices[6];
    matrix SkinNormalMatrices[6];
};

cbuffer LightParameters
{
    float3 LightPositionWS;
    float4 LightColor;
).J

Texture2D    ColorTexture  : register( t0 );
Texture2D    HeightTexture : register( t1 );
SamplerState LinearSampler : register( s0 );

struct VS_INPUT
{
    float3 position  : POSITION;
    int4   bone      : BONEIDS;
    float4 weights   : BONEWEIGHTS;
    float3 normal    : NORMAL;
    float2 tex       : TEXC00RDS;
};
//. ----------------------------------------------------------------
struct VS_OUTPUT
{
    float4 position  : SV_Position;
    float3 normal    : NORMAL;
    float3 light     ; LIGHT;
    float2 tex       : TEXC00RDS;
};
//--. . . . . . . . . . . . .--. ----------------------------------------
Struct HS_P0INT_0UTPUT
{
    float4 position  : SV_Position;
    float3 normal    : NORMAL;
    float3 light     : LIGHT;
    float2 tex       : TEXC00RDS;
};
//-----------------------------------------------------------------
struct HS_PATCH_0UTPUT

{
    float Edges[3]   : SV_TessFactor;
    float Inside     : SV_InsideTessFactor;
}i
```

```
//-----------------------------------------------------------------------------
struct DS_OUTPUT
{
    float4 position  : SV_Position;
    float3 normal    : NORMAL;
    float3 light     : LIGHT;
    float2 tex       : TEXCOORDS;
};
//-----------------------------------------------------------------------------
VS_OUTPUT VSMAIN( in VS_INPUT input )
{
    VS_OUTPUT output;

    // Calculate the output position of the vertex
    output.position  = (mul( float4( input.position, l.ef ),
      SkinMatrices[input.bone.x] )
                        * input.weights.x);

    output.position += (mul( float4( input.position, l.ef ),
      SkinMatrices[input.bone.y] )
                        * input.weights.y);
    output.position += (mul( float4( input.position, l.ef ),
      SkinMatrices[input.bone.z] )
                        * input.weights.z);
    output.position += (mul( float4( input.position, l.ef ),
      SkinMatricesfinput.bone.w] )
                        * input.weights.w);

    // Calculate the world space normal vector
    output.normal  = (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.x] )
                      * input.weights.x).xyz;
    output.normal += (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.y] )
                      * input.weights.y).xyz;
    output.normal += (mul( input.normal, (float3x3)SkinNormalMatrices[input.
      bone.z] )
                      * input.weights.z).xyz;
    output.normal += (mul( input.normal, (float3x3)SkinNormalMatrices[input,
      bone.w] )
                      * input.weights.w).xyz;

    // Calculate the world space light vector
    output.light = LightPositionWS - output.position.xyz;

    // Pass the texture coordinates through
    output.tex = input.tex;

    return output;
}
//-----------------------------------------------------------------------------
HS_PATCH_OUTPUT HSPATCH( InputPatch<VS_OUTPUT, 3> ip, uint PatchID : SV_PrimitiveID )
{
    HS_PATCH_OUTPUT output;
    const float factor = 16.0f;

    output.Edges[0] = factor;
    output.Edges[1] = factor;
    output.Edges[2] = factor;

    output.Inside = factor;

    return output;
}
//-----------------------------------------------------------------------------
[domain("tri")]
[partitioning("fractional_even")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("HSPATCH")]
HS_POINT_OUTPUT HSMAIN( InputPatch<VS_OUTPUT, 3> ip,
                        uint i : SV_OutputControlPointID,
                        uint PatchID : SV_PritnitiveID )
{
    HS_POINT_OUTPUT output;

    // Insert code to compute Output here,
    output.position = ip[i].position;
    output.normal = ip[i].normal;
    output.light = ip[i].light;
    output.tex = ip[i].tex;

    return output;
}
```

Listing 8.4. The vertex and hull shader programs and the patch constant function for setting up the tessellation process.

After the hull shader, the tessellator stage consumes the tessellation factors and generates a set of barycentric coordinates that will be passed to the domain shader. The domain shader also receives the control points that were passed from the hull shader main function, and must convert the barycentric coordinates into clip-space vertices. This is done by first calculating the world-space position of the barycentric point by interpolating the input control point positions. These are the world-space skinned-vertex positions that we calculated in the vertex shader, meaning that if we passed these vertices out with these interpolated positions, we would effectively generate a triangle mesh that appears the same as our normal skinned mesh. In addition to interpolating the position, we also interpolate the per-vertex normal vector, as well as the texture coordinates of the vertices, and renormalize the normal vector afterward to ensure that it remains a unit length vector. This process can be seen in Listing 8.5 which shows the complete domain shader program.

```
[domain("tri")]
DS_OUTPUT DSMAIN( const OutputPatch<HS_POINT_OUTPUT, 3> TriPatch,
                        float3 Coords : SVOomainLocation,
                        HS_PATCH_OUTPUT input )
{
    DS_OUTPUT output;

    // Interpolate world space position
    float4 vWorldPos = Coords.x * TriPatch[6].position
                     + Coords.y * TriPatch[l].position
                     + Coords.z * TriPatch[2].position;

    // Calculate the interpolated normal vector
    output.normal = Coords.x * TriPatch[0].normal
                  + Coords.y * TriPatch[l].normal
                  + Coords.z * TriPatch[2].normal;

    // Normalize the vector length for use in displacement
    output.normal = normalize( output.normal );

    // Interpolate the texture coordinates
    output.tex = Coords.x * TriPatch[0].tex
               + Coords.y * TriPatch[l].tex
               + Coords.z * TriPatch[2].tex;

    // Calculate the interpolated world space light vector.
    output.light  = Coords.x * TriPatch[0].light
                  + Coords.y * TriPatch[l].light
                  + Coords.z * TriPatch[2].light;

    // Calculate MIP level to fetch normal from
    float fHeightMapMIPLevel =
      clamp( ( distance( vWorldPos.xyz, vEye.xyz ) - 100.0f ) / 100.0f,
             0.0f,
             3.0f);

    // Sample the height map to know how much to displace the surface by
    float4 texHeight =
           HeightTexture.SampleLevel( LinearSampler, output.tex,
fHeightMapMIPLevel );

    // Perform the displacement. The 'fScale' parameter determines the maximum
    // world space offset that can be applied to the surface. The displacement
    // is performed along the interpolated vertex normal vector.
    const float fScale = 0.5f;
    vWorldPos.xyz = vWorldPos.xyz + output.normal * texHeight.r * fScale;

    // Transform world position with viewprojection matrix
    output.position = mul( vWorldPos, ViewProjMatrix );

    return output;
}
```

Listing 8.5. The domain shader program for implementing displacement mapping.

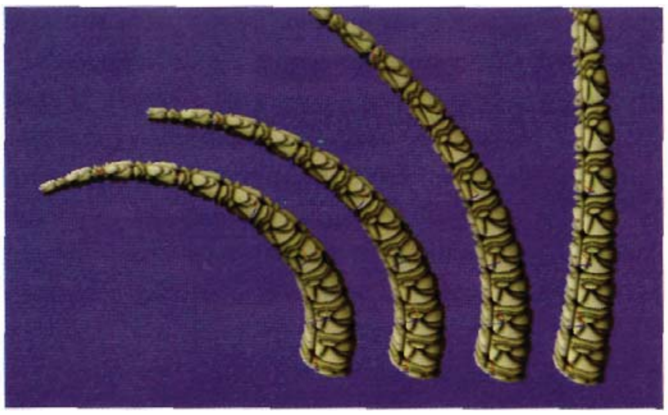

Figure 8.19. The results of using displacement mapping for mesh rendering.

With the interpolated world-space position, vertex normal vector, and texture coordinate, we can move ahead with the displacement mapping process. We first sample the displacement map (HeightTexture) to determine the desired magnitude of the displacement. This sampling process must use the SampleLevel function, since the domain shader stage can't automatically determine what mip-map level to use. Here we perform a simple calculation based on the distance from the viewer, then clamp the result between levels 0 and 3. The sampled height value is scaled by a constant after being read to allow for adjustments of the overall range of displacement values, if desired. The displacement magnitude is then used to scale the normal vector, and the result is added to the interpolated world-space position. The result of this calculation is the final output world-space position of the tessellated point, which represents a piece of our injected higher-resolution geometry.

Finally, the domain shader transforms the new vertex to clip space with the ViewProjMatrix transform and passes the texture coordinate to the pixel shader. As stated earlier, the pixel shader remains the same as in our earlier examples. From the point of view of the pixel shader, the geometry could have come from either the input vertex buffer or from the tessellation system—both would appear identical to the pixel shader. The resulting rendering contains higher-detail geometry where the user can see it, and can reduce the level of detail of the geometry where the user can't see it. Several sample renderings using the new algorithm are shown in Figure 8.19.

8.33 Conclusion

To review the algorithm we have implemented, we can now perform detailed animation of complex objects, while also reducing the amount of geometry that must be processed to obtain an equal amount of surface detail. Displacement mapping allows the expensive skinning calculations to be done at a lower detail level, but it still introduces high detail back in after skinning. Using the tessellation stages lets us ramp up the detail according to whatever metric we choose, which also provides a simple way to scale performance according to the hardware currently being used. Using a displacement map lets us obtain our additional geometric complexity relatively quickly with a single texture sample, and it also requires a relatively small amount of memory in comparison to an equivalent number of vertices. Finally, using a displacement map also allows for very simple use of LOD techniques to restrict the detail level returned by the sampled displacement map.