# The Rendering Pipeline

The *Direct3D 11 Rendering Pipeline* is the mechanism used to process memory resources into a rendered image with the GPU. The pipeline itself is made up of a number of smaller logical units, called *pipeline stages.* Data is processed by progressing through the pipeline one stage at a time and is manipulated in some way at each stage. By understanding how the individual stages of the pipeline operate, and the semantics for using them, we can use the pipeline as a whole to implement a wide variety of algorithms that execute in real time.

As GPUs have become increasingly powerful with each new architecture generation, the size and capabilities of the pipeline have expanded significantly. In addition, the complexity and configurability of each pipeline stage has steadily increased. The current rendering pipeline has fixed-function stages, as well as programmable shader stages. This chapter will first consider the differences between these types of pipeline stages. In particular, it will focus on the states that are required to make them function, as well as the processing they can perform. After clarifying this distinction, we will consider the higher-level details of how the pipeline is invoked, and how each pipeline stage communicates with its neighbors.

We will then explore each of the pipeline stages in detail. This includes the individual functions that each stage performs, a thorough discussion of how the stages are configured, and the general semantics that they bring along with them. With a firm understanding of what each individual component of the pipeline does, we can consider some of the higher-level functionalities that are implemented by groups of pipeline stages in various configurations. We will also discuss several high-level data processing concepts for the pipeline as a whole, including how to manage such a complex processing architecture. The rendering pipeline has evolved into a sophisticated set of APIs, which can be used to implement a wide variety of algorithms. After completing this chapter, we will have a deep and thorough understanding of how to use the rendering pipeline to develop efficient and interesting rendering techniques for use in real-time rendering applications.

# 3.1 Pipeline State

To understand how the pipeline operates, we need to look no further than its name. Data is submitted as input at one end of the pipeline, and then processed by the first pipeline stage. This data consists of vector-based variables with up to four components. After processing in the first stage is completed, the modified output data is passed on to the next stage. The next set of data is then brought into the first stage. This means that the first two stages are processing different pieces of data at the same time. This process is repeated until the complete pipeline is operating simultaneously on different portions of the input data. The pipeline architecture specifically allows multiple operations to be performed at the same time by different pipeline stages, which lets many specialized processes be carried out on an individual data item as it travels through the pipeline. Once a data item reaches the end of the pipeline, it is stored to an output resource, which can later be used as needed by the host application. This pipeline concept is a simple but powerful processing technique, similar in nature to an assembly line. Figure 3.1 shows how data is processed by a pipeline.

The task of the developer is to properly configure each stage of the pipeline to obtain the desired result when the data emerges from the end of the pipeline. The pipeline configuration process is performed by manipulating the state of each individual stage of the pipeline. By organizing the pipeline into stages, Direct3D 11 effectively groups related sets of states together and consolidates how they are manipulated. There are two different types of pipeline stages, the fixed function stages and the programmable shader stages. Both stage types share some common concepts regarding how data flows through them and how their states are manipulated by an application. The following sections will explore these two types of states and provide some general concepts of how to work with them.

## 3.1.1 Fixed Pipeline Stages

The *fixed-function pipeline* stages perform a fixed set of operations on the data passed to them. They perform specific operations, and hence provide a "fixed" scope of available functionality. They can be configured in various ways, but they always perform the same

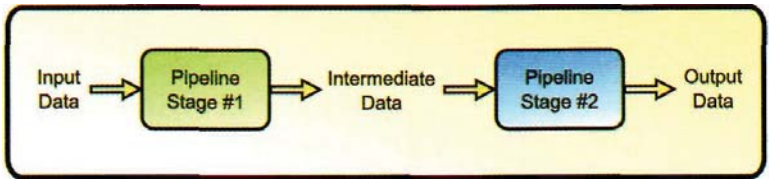



Figure 3.1. The Direct3D 11 rendering pipeline.

operations on data passed to them. A useful analogy to this concept is to consider how a regular function works in C++. You first pass the function a predefined list of parameters. It then processes the data and returns the result as its output. The body of the function represents the operations that the fixed function stage implements, while the input parameters are the available configurations and the actual input data. We can't change the function body, but we can control what options are used during processing of the input data.

Some examples of this type of fixed functions in previous generations of Direct3D include the ability to change the vertex culling order (which is now a portion of the rasterizer stage), selecting a depth test function (now a portion of the output merger stage), and setting the alpha blending mode (also a portion of the output merger stage). This focus on a single functional area is primarily due to performance considerations. If a particular pipeline stage is designed for a specific task, it can usually be optimized to perform that task more efficiently than a general purpose solution could.

In Direct3D 9, changing the state of these fixed function stages was performed by calling an API function for each individual setting that was to be changed. This required many API calls to be performed to configure a particular processing setup for each stage. Depending on the scene contents and the rendering configuration, the number of API calls could easily add up and begin to cause performance problems. Direct3D 10 introduced the concept *of state objects* to replace these individual state settings. A state object is used to configure a complete functional state with a single API call. This significantly reduces the number of API calls required to configure a state and also reduces the amount of error checking required by the runtime. Direct3D 11 follows this state object paradigm. The application must describe the desired state with a description structure, and then create a state object from it, which can be used to control the fixed-function pipeline stages.

The requirement to create a complete state all at once moves state validation from the runtime API calls to the state creation methods. If incompatible states are configured together, an error is returned at creation time. After creation, the state object is immutable and cannot be modified. Thus, there is no way to set an invalid state for the fixed function pipelines, which effectively removes the validation burden from the state-setting API calls. Overall, this system of using state objects to represent the pipeline state allows for a more streamlined pipeline configuration, with only minimal application interactions required.

### 3.1.2 Programmable Pipeline Stages

The *programmable pipeline stages* comprise the remaining stages of the pipeline. The word programmable here means that these stages can execute programs written in the High Level Shading Language (HLSL). Using the same C++ analogy from above, the programmable stages allow you to define the required input parameters *and the* body of the function. In fact, the programs that run in these stages are authored as functions in HLSL.

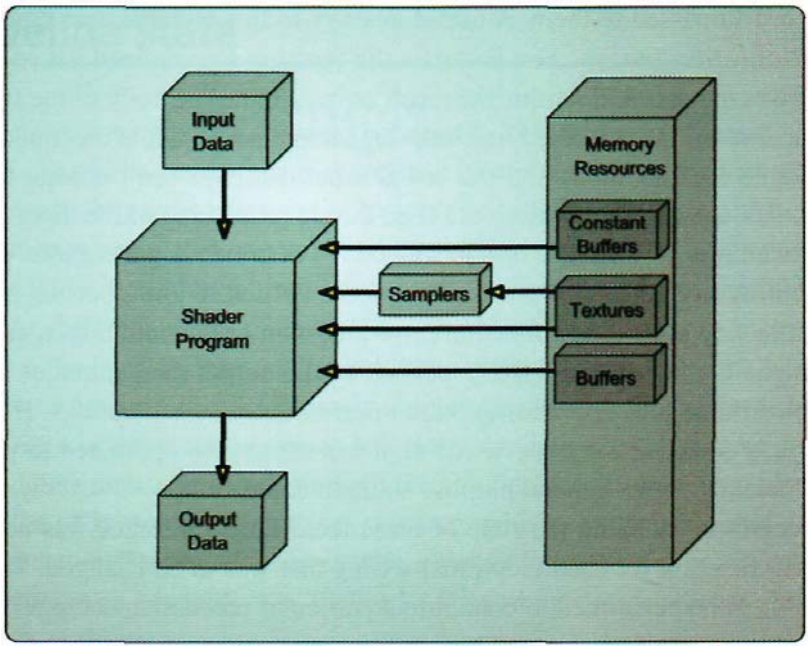



Figure 3,2. A block diagram of the common shader core.

The ability to execute a program lets these pipeline stages be used for a wide variety of processing tasks. This contrasts with the fixed function stages, which are intended for a very specific task and only offer a small amount of configurability. The programs that are executed in these programmable pipeline stages are commonly referred to as *shader programs,* a name inherited from the early steps of programmability, when pixel shader stage was initially used to modify how an object reacted to lighting. As more and more programmable stages were added to the pipeline, the name *shader* was adopted to refer to all of the programmable pipeline stages. We will use this term interchangeably with *programmable stages* throughout the book. In the following sections, we will first take a look at these programmable shader stages from a high level, and then dive into the details of their architecture, after establishing the basic concepts.

## Common Shader Core

All of the programmable shader stages are built upon a common base of functionality, which is referred to as the *common shader core.* The common shader core defines the generic input and output design of the pipeline stage, providing a set of intrinsic functions that all of the programmable shader stages support, as well as the interface to the resources that a programmable shader can use. Figure 3.2 provides a visual representation of how the common shader core operates. As described above, the data flows into the top of the stage, is processed within the shader core, and then flows out as output at

the bottom of the stage. The shader program that executes within these stages is a function written in HLSL for a specific purpose. While the data is being processed within the shader stage, the shader program has access to constant buffers, samplers, and shader resource views that are bound to that stage by the application. Once the shader program has finished processing its data, that data is passed out of the stage, and the next piece of data is brought in to start the process over again.

The configurability of the programmable pipeline stages is not limited to the processing performed within the shader programs. The input and output data structures for these stages are also specified by the shader program, which provides a flexible way to pass data from stage to stage. Some rules are imposed on these interfaces between stages, such as requiring certain types of data to be included in the interfaces of particular stages. A typical example of a required output is for one of the stages before the rasterizer to provide a position output that can be used to determine which pixels of a render target are covered by a primitive. Depending on which stages are passing the data, the parameters are either provided unmodified, or can be interpolated before being passed on to the next stage.

While all the programmable stages share a common set of functionality, each stage can also provide additional specialized features and functionalities unique to it. These are usually related to input and output semantics, and are hence only applicable to particular stages. The individual behaviors of each stage will be discussed in more detail later in this chapter as we describe each stage of the pipeline.

With so much flexibility, and a wide variety of different pipeline stages, there are many algorithms that do not require all stages to be active. Thus, it is possible to disable the programmable shader stage's by clearing its shader program. In addition, since there is a large amount of flexibility in the processing done in each stage, it is quite possible to perform some or all of the required work in different combinations of all of the programmable stages. This ability to choose where to perform a particular calculation can be used to our advantage. If a calculation can be done prior to any data amplification, it is possible to calculate the same data with far fewer operations. We will see a good example of this in the Chapter 8, "Mesh Rendering," where vertex skinning is performed prior to a model being tessellated, to reduce the number of vertices being skinned. (The details of what vertex skinning is can be found in Chapter 8.)

### Shader Core Architecture

In the previous section, we have seen the general concept of how a programmable stage operates. It receives input from the previous stage, executes an HLSL program on it, and passes the results on to the next pipeline stage. However, this is just an overview description of what is really going on. The shader program that executes in a programmable stage is actually compiled from HLSL into a vector-register-based assembly language designed for use in specialized shader processor cores in the GPU. Even though all shader programs

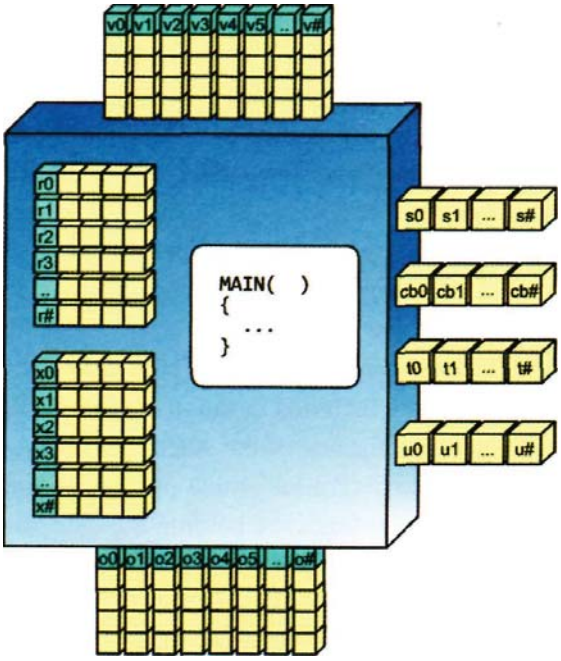



Figure 3.3. The assembly language view of the common shader core.

must be written in HLSL, they must still be compiled into this assembly byte code before being used in the rendering pipeline.[12]

We can learn a great deal of information from this assembly language. It defines a specific set of registers that can be used by the compiler to map an HLSL program to assembly language. The registers are generally four component vector registers, which can use individual components to provide scalar register functionality as well. There are registers for a shader core to receive its input data, temporary registers for performing computations, registers for interacting with resources, and registers for passing data out of the stage. The assembly language instructions use these registers to perform their respective operations. By understanding how the assembly programs can use these registers, we can gain a deep insight into how shader stages operate.

To get started, we will consider the common shader core overview shown in Figure 3.2, but this time we will view it from an assembly language viewpoint. Figure 3.3 shows the assembly version of the common shader core.

[1] The assembly program produced by compiling an HLSL program is not directly executed in the GPU. It is further processed by the video driver into machine-specific instructions, which can vary from GPU to GPU. Even so, the assembly language provides a common point of reference with which we can gain an insight into the operations of a shader processor.

[2] The details of the compilation process can be found in Chapter 6, "The High Level Shading Language.'

As seen in Figure 3.3, the input to the shader core is provided in the v# registers.[3] Since they are providing the input to the stage, they are naturally read only. When a shader program is executed, its input data is available in the v# registers. After the data has been read, it can be manipulated and combined with other data, and any intermediate calculations can be stored in the r# and x#[n] registers. These are called *temporary registers,* and since they hold intermediate values, they are both readable and writable by a shader program. The texture registers (t#), constant buffer registers (cb#[n]), immediate constant buffer register (icb[index]), and unordered access registers (u#) are also available as data sources. These registers are used to provide access to the device memory resources, as described in Chapter 2, and they are all read only except for the unordered access registers. Finally, the calculated values that will be passed on to the next pipeline stage are written into the output registers (o#). When the shader program has terminated, the values stored in the output registers are passed on to the input registers of the next stage, where the process is repeated. A few other special purpose registers are only available in certain stages, so we will defer discussion of them until later in the chapter.

Typically, a developer does not need to inspect the assembly listing of a compiled shader program, unless there is a performance issue.[4] This makes understanding the details of how the assembly instructions operate less critical. Even so, it is still helpful to have a basic knowledge of the assembly-based world. For example, when developers define input and output data structures for a shader program, they must be aware of the limits on how many input and output vectors can be used for each stage. This is determined by the number of input and output registers available for that particular stage. Similarly, the available number of constant buffers, textures, and unordered access resources is limited by their respective registers, as well. This is very important information, and should be taken into consideration as we proceed through each of the pipeline stage discussions.

### GPU Architectures

Even with a strictly defined assembly language specification, the actual GPU hardware is not required to directly implement the specification. There are many different architectural implementations, which can vary widely from one vendor to the next. In fact, even consecutive generations of GPU hardware from the same vendor can vary significantly from one another. This makes it incredibly difficult to predict how efficiently a given shader program will execute on a current or future GPU. Depending on the architecture of the GPU executing the program, one particular memory access pattern may be more efficient than another, but the opposite could be true with a different architecture.

[3] The # symbol indicates that multiple registers are available that are identified by an integer index. For example, v0 and vl are the first two input registers available for use.

[4] Details about how to compile a shader and view its assembly listing are provided in Chapter 6.

Due to the huge amount of variation among implementations, it would be impractical to provide detailed information here. We invite the reader to explore this fascinating topic in more detail, with a good starting point available in (Fatahalian). However, we can still provide a general idea of how a GPU is organized, so that discussions later in the book will have a context to build upon. The GPU has evolved into a massively parallel processor, housing hundreds of individual ALU processing cores. These processors can run custom programs, and they can access a large bank of memory, which allows for high-bandwidth data transfers. Each GPU uses some form of a memory cache system to reduce the effective latency of memory requests, although the cache system is vendor specific and its low-level design is generally not disclosed. In the past, it has been necessary to understand the individual architectural details of a particular target GPU in order to attain the maximum available performance level. This trend will likely continue for some time into the future, due to the fact that GPUs are still evolving at a very rapid pace.

## 3.2 Pipeline Execution

The fixed function stages, together with the programmable stages, combine to provide an interesting and diverse pipeline with which we can render images. The process of executing the pipeline consists of configuring all of the pipeline stage states, binding input and output resources to the pipeline, and then calling one of the Draw methods to begin execution. All of these tasks are performed through the methods of the ID3DllDeviceContext interface. The number of pipeline executions performed to generate a rendered frame depends on the application and the current scene, but we can assume that at least one pipeline execution is performed for each frame.

Once the pipeline has been invoked with one of these draw methods, data is read into the first stage of the pipeline from the input memory resources. The amount of data read from the input resources depends on the type and parameters of the draw call used to invoke the pipeline. Each piece of data is processed and passed on to the next stage until it reaches the end of the pipeline, where it will be written to an output memory resource. Once all of the data from these draw calls have been processed, the resulting output resource (typically a 2D render target) can be presented to an output window or saved to a file.

Throughout this chapter, we will see how to configure each stage of the pipeline, as well as understand the differences between each of the draw methods. There are seven different draw methods that can be used to invoke the rendering pipeline. They are listed here for easy reference, and to demonstrate that a fairly wide variety of methods is available for invoking the rendering pipeline.

```
Draw(...)

DrawAuto(...)

DrawIndexed(...)

DrawIndexedInstanced(...)

DrawInstanced(...)

DrawIndexedInstancedIndirect(...)

DrawInstancedIndirect(...)
```

Each of these methods instructs the pipeline to interpret its input data in a different way. Each method also offers different parameters to further configure how much, and which, input data to process. Once the pipeline execution has been initiated, the input data specified in the draw method is processed piece by piece as it is brought into the pipeline.

One of the most important considerations when planning the work that will be performed during one pipeline execution is that each of the stages (both fixed and programmable) has different input and output types. In addition, the semantics for using those types have relatively large performance implications. For example, in a simple rendering sequence we could submit four vertices to the pipeline as two triangles. This would lead to four vertex shader invocations, which would pass their output to the rasterizer stage, which produces a number of fragments. These fragments are then passed to the pixel shader, then finally to the output merger. If the triangles are close to the viewer, many fragments may be generated, causing correspondingly large number of pixel shader invocations. The number of vertices is in this case is constant, while the number of fragments depends on the conditions of the scene. The opposite situation can also occur, where the tessellation stages produce a variable number of vertices to be rasterized, but the object remains the same size onscreen, and thus the number of pixels remains constant. One must consider this when designing an algorithm, since it is critical to balancing the workload placed on the GPU.

During the actual execution of the pipeline on hardware, each individual pipeline stage must perform its calculations as quickly as possible on the available GPU hardware. In many cases the GPU can perform multiple types of computation at the same time in different groups of processors. In the example above, it would be possible for the GPU to process the first triangle and rasterize it to generate fragments. It could then use some of its processing cores to process pixel shader invocations, and some to perform the remaining vertex shader work. On the other hand, it could process all of the vertices first and then switch to processing all of the fragments afterward. There is no guarantee about which order the GPU will process the data in—except that it will respect the logical order presented in the pipeline.

### 3.2.1 Stage-to-Stage Communication

As data is processed by the individual pipeline stages, it must be transmitted from one stage to the next. To do this, each programmable stage defines its required inputs and outputs for its shader program. We will refer to these inputs and outputs as *attributes,* which consist of vector variables (from 1 to 4 elements). All of the elements in each vector consist of one of the scalar types (both integer and floating point types are available). We can also refer to these attributes as *semantics* or *binding semantics,* which is the name given to the text-based identifier for each input/output attribute. In general, it should be clear from the context in which they are used when we are referring to these types of stage-to-stage variables by either of these names.

The input attributes to a shader program define the information required for that program to execute. Quite literally, the input attributes are the input parameters to the shader program function. Likewise, the function's return value defines its output attributes. Thus, each previous stage must produce output parameters that match the required input parameters for the next stage. These input and output attributes are implemented with the input and output registers we discussed in the "Shader Core Architecture" section. The brief example in Listing 3.1 provides a sample declaration for two shader functions, with each function defining its inputs and outputs. Notice that the output from the first function matches the input from the second function.

```
struct VS_INPUT
{
    float3 position : POSITION;
    float2 coords : TEXCOORDS;
    uint vertexID : SV_VertexID;
};

Struct VS_OUTPUT
{
    float4 position : SV_POSITION;
    float4 color : COLOR;
};

VS_OUTPUT VSMAIN( in VS_INPUT v )
{
    // Perform vertex processing here...
}

float4 PSMAIN( in VSJXJTPUT input ) : SV_Target
{
    // Perform pixel processing here...
}
```

Listing 3.1. Two shader functions defining their input and output attributes which must match.

In the declaration of each function's input and output, a structure definition is used to group all of the inputs and outputs together. Each attribute of the input and output structures has an associated semantic, a textual name to link two data items together between stages. The semantic is located after the name of the attribute, following a semicolon. You can also see the type declarations for each of the components to the left of their names, just as you would expect from a similar declaration in C/C++.

You can see from this example that the output from the first shader function nearly matches the input of the second function, with the exception of the vertexID parameter. The semantic listed next to this particular parameter is SV_VertexID. In addition to the user-defined semantic attributes shown in Listing 3.1, there is another group of parameters that can be used by a programmable shader in its input/output signature. These are referred to as *system value semantics.* They represent attributes that the runtime either generates or consumes, depending on where they are declared, and which system value semantics are being used. These system value semantics provide helpful information for use in HLSL programs, and are also used to indicate required values to the pipeline. System values are always prefixed with SV_ to indicate that they have a special significance and are not user defined. We will see each of the system values, and how they are used in each of the pipeline stages, as we progress through the pipeline.

Understanding how data is supplied to the pipeline, and seeing how each pipeline stage's states influence the type of processing that occurs, are the key to successfully using the rendering pipeline. In the following sections we will inspect each of the pipeline stages in great detail, in the order in which they appear within the pipeline.

## 3.3 Input Assembler

The *input assembler* stage is the first stop in the rendering pipeline. It is a fixed function stage that is responsible for putting together all of the vertices that will be processed further down the pipeline—hence the name. As the entry point of the pipeline, the input assembler must create vertices that have the attributes required in the next pipeline stage, the vertex shader. The process of assembling vertices from one or more vertex buffer resources can actually cover a large number of different configurations, as we will see in detail later in this section. The application provides the input assembler a road map to constructing the vertices with a vertex layout object, which also allows for flexible strategies on the application side when creating vertex buffer resources. The input assembler's location in the pipeline is highlighted in Figure 3.4.

In addition to constructing the input vertices, the input assembler also determines how those vertices are connected to one another by specifying the topology of the geometry being rendered. This identifies the types of primitives or control points that define

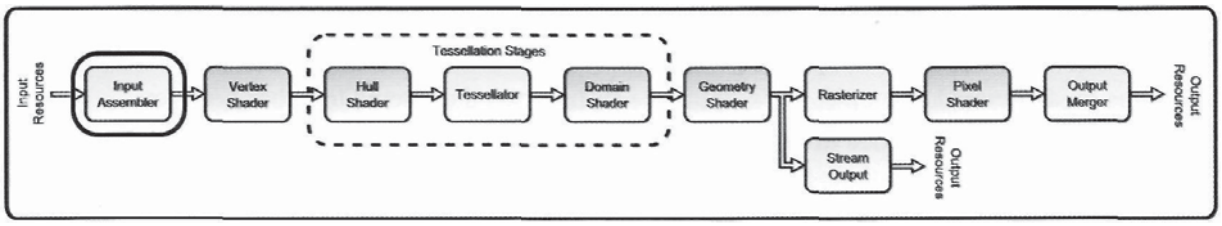



figure 3.4. The input assembler stage.

how the individual vertices should be grouped together later in the pipeline. By specifying the topology of the geometry being rendered, the input assembler can have a significant impact on how the same set of input vertices is interpreted by the rest of the pipeline. If a geometric object is composed of triangles, but it is rendered as a set of points, the resulting rendering will be quite different, even though the input vertex buffers are exactly the same.

The input assembler is also the enabler that allows several different types of rendering operations to be performed by the rendering pipeline. The rendering pipeline supports standard draw calls, indexed drawing, instanced drawing, indirect drawing, and several combinations of these various operations—all of which require different input resource configurations. The input assembler works in conjunction with its input resources and the information provided in the draw call to convert these various input configurations into a format that is usable by the rest of the pipeline. In many cases, the input assembler is the only stage that executes differently for different draw calls. It effectively hides the data submission details from the rest of the pipeline, and instead, provides its output data in a consistent format. Since the input assembler forms the primary link between an application's model data and the rendering pipeline, it serves a very important function. We will investigate this pipeline stage in more detail to understand precisely how to properly use it, and what options are available for invoking a pipeline execution.

## 3.3.1 Input Assembler Pipeline Input

Prior to passing data into the pipeline, the input assembler must be connected to the appropriate resources to acquire the input geometric data. This data consists of vertex data and index data, both of which are supplied to the pipeline in the form of buffer resources. The following two sections describe how to bind these buffer resources to the input assembler.

### Vertex Buffer Usage

Having primary responsibility to correctly assemble vertices for use further down the pipeline, the input assembler requires access to vertex data from the application. This data is stored in one or more vertex buffer resources and can be organized in a variety of different

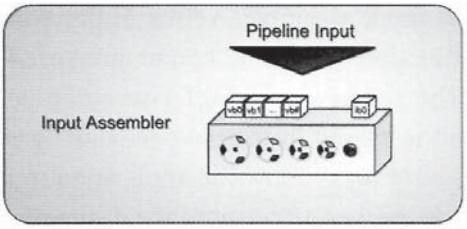



Figure 3,5. An overview of the available input slots of the input assembler.

layouts. Conceptually, we can view the input assembler as having several input slots, which can be filled with vertex buffer resources. Figure 3.5 shows this visualization.

Each slot can be filled with a vertex buffer resource that contains one or more attributes of a complete vertex. For example, one vertex buffer can contain the position data for the vertices, and a second vertex buffer can contain all of the normal vector data. It is also possible to provide all of the vertex data within a single vertex buffer in an array of structures. The developer can choose how to best organize the input data. There are currently 16 input slots available for the application to configure, which allows for fairly flexible storage options for the needed vertex data. With the individual attributes stored separately, it is possible to dynamically decide which vertex components will be needed for a particular rendering. This can potentially reduce the bandwidth required for reading the vertex data, by eliminating unused attributes from the vertices. We will discuss the details of how to store various data in multiple buffers later in this chapter.

To bind the vertex buffers to the input assembler stage, we use the device context interface. As we saw in Chapter 1, the device context is the primary interface to the complete pipeline. The process to bind several vertex buffers to the input assembler with the ID3D11 DeviceContext:: IASetVertexBuff ers method is shown in Listing 3.2. Each of the variables declared in the listing will be filled with its appropriate buffer references, strides, and offsets, before the set vertex buffer method is called.

```
UINT StartSlot;
UINT NumBuffers;
ID3DIIBuffer*   aBuffers[D3D11_IA_VERTEX_INPUT_RES0URCE_SL0T_C0UNT];
UINT aStrides[D3DII_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT];
UINT a0ffsets[D3DII_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT];

// Fill in all data here...

pContext->IASetVertexBuffers( StartSlot, NumBuffers., aBuffers, aStrides,
                              aOffsets );
```

Listing 3.2. How to bind several vertex buffers to the input assembler stage.

The index of the first input assembler vertex buffer slot to begin binding to, and how many subsequent buffer slots should be bound, are specified in the StartSlot and NumBuffers parameters. The list of vertex buffer resource pointers must be assembled into a contiguous array, and is passed in the ppVertexBuffers parameter. The final two parameters, pStrides and pOffsets, allow the application to specify the per-vertex step size for each vertex buffer, as well as an offset to the desired location to begin using each vertex buffer. Each of these array elements will start being used at the StartSlot index and will be used to fill NumBuff ens number of slots. This means that the arrays must be appropriately sized and filled in, or the function will access an invalid memory location. In addition, to bind the vertex buffers to the input assembler, the buffers must not be bound for writing at another location in the pipeline. This is a reasonable requirement, since it would lead to seemingly unpredictable results, with buffers being read from and written to at the same time.

### Index Buffer Usage

When taken on its own, vertex data is interpreted by the input assembler in the order that it appears within the vertex buffers. This means that if the vertex data is being used to render triangles, the first three vertices define the first triangle, followed by the next three vertices for the next triangle, and so on.[5] This can lead to duplicated vertex data in cases where two adjacent triangles actually reference the same vertex. In fact, this is usually the case—most triangle meshes have a significant number of adjacent triangles. When the individual vertex data is specified more than once in the vertex buffer, significant amounts of memory can be wasted. This inefficiency can be eliminated with the use of index buffers. An index buffer is used to define the vertex order for each primitive by providing index offsets into the vertex buffers. In this case, each vertex can be specified once, and then an index that points to it can be used for any triangle that it is a part of. With indexed rendering, the three vertices to be used in a triangle are specified by three consecutive indices in the index buffer.

To use indexed rendering, an application must bind an index buffer to the input assembler. There is only one slot available for binding an index buffer, and it is only used during one of the indexed draw calls to specify the order of the vertices used to generate primitives. To bind the index buffer to the input assembler, the application must use the ID3DllDeviceContext: :IASetIndexBuffer method. Listing 3.3 demonstrates how to perform this binding operation. This method provides three simple parameters: a pointer to the index buffer, the format that the indices are stored in (either 16- or 32-bit unsigned integers), and finally an offset into the index buffer to begin building primitives from.

[5] The ordering of vertices and indices will vary for different primitive types. These are described in more detail later in this section.

```
ID3D11Buffer* pBuffer = 0;
UINT offset = 0;

// Set pBuffer to the desired index buffer reference here, and
// set the offset to an appropriate location in the buffer.

m_pContext->IASetIndexBuffer( pBuffer, DXGI_FORMAT_R32_UINT, offset );
```

Listing 3.3. How to bind an index buffer to the input assembler stage.

Once a vertex or index buffer is bound to the pipeline, it can be unbound by setting a NULL pointer value to the corresponding input slot. This is important to remember, especially when reconfiguring the pipeline in between draw calls. If a multiple vertex buffer configuration is used in a previous draw call, and the new draw call uses fewer vertex buffers, the unused slots should be filled with NULL to unbind the unused buffers. This is why the vertex buffer binding method always uses an array sized for the maximum number of vertex buffers and initializes them to NULL values before filling the desired configuration.

## 3.3.2 Input Assembler State Configuration

After the appropriate resources have been bound to the vertex and index buffer slots, there are two other configurations that must be set in the input assembler before it can be used. The first is the Input Layout object, which is used by the input assembler to know which input slots to read the per-vertex data from to build complete vertices. The second parameter is the primitive topology that should be used by the input assembler to determine how vertices are grouped together into primitives. These two states, in combination with the draw method that is used to execute the pipeline, determines how vertex and primitive data are interpreted by the pipeline. We will explore each of these states in detail and then consider how they interact with the available draw methods to produce different input configurations.

### Input Layout

The Input Layout object can be thought of as a recipe that tells the input assembler how to create vertices. Every vertex is composed of a collection of vector attributes, each with up to four components. With up to 16 vertex buffers available for binding, the input assembler needs to know where to read each of these components from, as well as understanding what order to put them in the final assembled vertices. The Input Layout object provides this information to the input assembler.

To create an Input Layout object, the application must create an array of D3D11_INPUT_ELEMENT_DESC structures, with one structure for each component of a desired vertex

configuration. The members of D3D11_INPUT_ELEMENT_DESC are shown in Listing 3.4. We will examine what each of these structure members represents, and how they define the needed information for the input assembler to do its job.

```
Struct D3D11_INPUT_ELEMENT_DESC {
    LPCSTR                      SemanticName;
    UINT                        SemanticIndex;
    DXGI_FORMAT                 Format;
    UINT                        InputSlot;
    UINT                        AlignedByteOffset;
    D3D11_INPUT_CLASSIFICATION  InputSlotClass;
    UINT                        InstanceDataStepRate;
}
```

Listing 3.4. The D3D11_INPUT_ELEMENT_DESC structure members.

**Per-vertex elements.** The first parameter, SemanticName, identifies the textual name of a vertex attribute, which must match the corresponding name provided in the vertex shader program. Each input to a vertex shader must have a semantic defined in its HLSL source code. That semantic name is used to match the vertex shader input to the vertex data provided by the input assembler. The SemanticIndex is an integer number which allows a SemanticName to be used more than once. For example, if more than one set of texture coordinates is used within a single vertex layout, each set of coordinates could use the same SemanticName, but would use increasing SemanticIndex values to differentiate between them.

The next structure member is the Format of the component. This specifies what data type, and how many elements are contained in the attribute. The available formats range from 1 to 4 elements, with both floating point and integer types available. The next parameter is the InputSlot, which indicates which of the 16 vertex buffer slots this component's data should be read from. The AlignedByteOffset indicates the offset to the first element in a vertex buffer for the items described by this D3D11_INPUT_ELEMENT_DESC. This tells the input assembler where in a vertex buffer to begin reading input data from.

**Per-instance elements.** The final two structure members declare vertex functionality regarding instancing. The instancing draw methods essentially use a single draw call to submit a model to the pipeline. This model is then "instanced" multiple times, but with a subset of the vertex data specified at a per-instance level, instead of a per-vertex level. This is frequently used when many copies of the same object must be rendered at different places throughout a scene. The vertex format will include a transformation matrix in its definition, which is only incremented to the next value for each instance of the model. Then the transformation matrices for all of the instances of the model are provided in a vertex

buffer and bound to the input assembler. When the draw call is performed, the vertex data is submitted to the vertex shader one copy at a time, with the transformation matrix being updated between instances of the model.

If a vertex component is providing per-instance data, it must specify that with the InputSlotClass parameter. In this case, there is also an option to increment to the next element in the vertex buffer only after a certain number of instances have been submitted to the pipeline. This is specified in the InstanceDataStepRate and provides a rough control for the frequency with which a per-instance parameter is changed. Since this is specified at the individual attribute level, it is possible to have different attributes to advance to the next data member at different rates. After all of the desired vertex attributes have been identified, and an array of description structures has been filled in, the application can create the ID3DllInputLayout object with the ID3DllDevice: :CreateInputl_ayout() method. Listing 3.5 demonstrates how this process is performed.

```
// Create array of elements here for the API call.
D3D11_INPUT_ELEMENT_DESC* pElements = new D3D11_INPUT_ELEMENT_DESC[elements.
      count()];

// Fill in the array of elements (this will vary depending on how they are
// submitted to this method).
for ( int i = 0; i < elements.count(); i++ )
    pElements[i] = elements[i]j;

// Attempt to create the input layout from the input information (the
// compiled shader byte code storage will also vary by the data
// structures used).
ID3DBlob* pCompiledShader = m_vShaders[ShaderID]->pCompiledShader;
ID3DIlInputtayout* pLayout = 0;

// Create the input layout object, and check the HRESULT afterwards.
HRESULT hr = m_pDevice->CreateInputLayout( pElements, elements.count(),
    pCompiledShader->GetBufferPointer(), pCompiledShader->GetBufferSize(),
                                         &pLayout );
```

Listing 3.5. The creation of an input layout object.

Here we can see the array of D3D11_INPUT_ELEMENT_DESC structures being allocated and then being filled in from an input container object. The first two parameters to ID3D11 Device: :CreateInputLayout() provide the array of descriptions and the number of elements that exist in the array. The third and fourth parameters are a pointer to the shader byte code that results from compiling the vertex shader source code, and the size of that byte code, respectively. This is used to compare the input array of element descriptions against the HLSL shader that it will be used to supply data to. Comparing these two objects

while creating the input layout lowers the amount of validation required at runtime when a shader is used with a particular input layout.

### Primitive Topology

The final state that must be set in the input assembler is the *primitive topology*. We can view the work of the input assembler as two tasks—first, it must assemble the vertices from the provided input resources, using the input layout object; and second, it must organize that vertex stream into a stream of primitives. Each primitive uses a small sequential group of vertices (with the sequence determined either by their order of appearance in the vertex buffers, or by the order of the indices in the index buffer for indexed rendering) to define the components that make it up. Common examples of primitive topology are triangle lists, triangle strips, line lists, and line strips. In the example of a triangle list, a primitive is created for every three vertices in the vertex stream.

The primitive topology setting tells the input assembler how to build the various primitives from the assembled vertex stream. The number of vertices used in each primitive, as well as how they are selected from the assembled vertex stream, is a function of the primitive topology setting. The available primitive types are provided in Listing 3.6, along with a demonstration of how to set the primitive topology with the ID3DllDeviceContext::IASetPrimitiveTopology() method.

```
enum D3D11_PRIMITIVE_T0P0L0GY {
    D3Dll_PRIMITIVE_T0POL0GY_UNDEFINED,
    D3D11_PRIMITIVE_T0P0L0GY_P0INTLIST,
    D3D11_PRIMITIVE_T0P0L0GY_LINELIST,
    D3D11_PRIMITIVE_T0P0L0GY_LINESTRIP,
    D3D11_PRIMITIVE_T0P0L0GY_TRIANGLELIST,
    D3D11_PRIMITIVE_T0P0L0GY_TRIANGLESTRIP,
    D3D11_PRIMITIVE_T0P0L0GY_LINELIST_ADJ,
    D3D11_PRIMITIVE_T0P0L0GY_LINESTRIP_ADJ,
    D3D11_PRIMITIVE_T0P0L0GY_TRIANGLELIST_ADJ,
    D3D11_PRIMITIVE_T0P0L0GY_TRIANGLESTRIP_ADJ,
    D3D11_PRIMITIVE_T0P0L0GY_1_C0NTR0L_P0INT_PATCHLIST,
    D3D11_PRIMITIVE_TOP0LOGY_2_CONTROL_POT.NT_PATCHLTST,
    D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_5_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_6_C0NTR0L_P0T.NT_PATCHLTST,
    D3D11_PRIMITIVE_T0P0L0GY_7_C0NTR0L_P0INT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_8_C0NTR0L_P0INT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_9_C0NTR0L_P0INT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_10_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_11_C0NTR0L_P0INT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_12_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_13_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_14_C0NTR0L_P0INT_PATCHLIST,
```

```
    D3Dll_PRIMITIVE_TOPOLOGY_15_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_16_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_17_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_18_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_19_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_20_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_21_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_22_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_23_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_T0P0L0GY_24_C0NTR0L_P0INT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_25_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_26_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_27_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_28_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_29_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_30_CONTROL_POINT_PATCHLIST,
    D3D11_PRIMITIVE_TOPOLOGY_31_CONTROL_POINT_PATCHLIST,
    D3Dll_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST
}

// Specify the type of geometry that we will be dealing with.
m_pContext->IASetPrimitiveTopology( primType );
```

Listing 3.6. The available primitive topology types, and an example of how to set the topology type with the device context interface.

As you can see from Listing 3.6, there are quite a variety of available primitive types. The first nine entries in the list should look familiar if you have used Direct3D 10 before, since they have been in use prior to Direct3D 11. However, the remainder of the list provides a series of different control point patch lists, each with a different number of points to be included in the control patch, up to a maximum of 32. These primitive types have been introduced to support the new tessellation stages that were introduced in Direct3D 11. To better understand how all of these primitive topologies organize a stream of vertices, we will create an example with an ordered series of vertices and then examine how each topology type would create primitives with this vertex stream. Figure 3.6 shows the sample set of vertices, each numbered according to its location in the vertex stream.

**Point primitives.** The *point list primitive topology* is the simplest of all of the primitive types. Vertices are grouped into single-vertex primitives, meaning that the stream of vertices produces an equal-sized stream of primitives. Therefore, the output group of primitives will look identical to that shown in Figure 3.6.

**Line primitives.** The *line list primitive topology* is only slightly more complex, with every pair of vertices producing a line primitive. This essentially indicates that the two vertices comprising a line must be included in the vertex buffer for every line to be rendered. The line strip primitive topology provides a more compact representation of the line primitives,

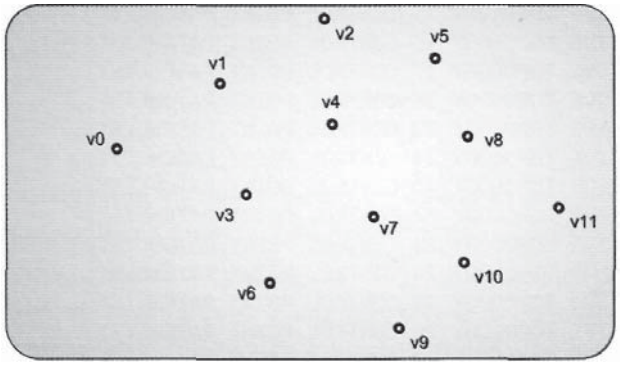



Figure 3.6. A number of vertices to be used to create a variety of primitive types.

but it can only represent a connected list of lines. The input assembler produces the first line from the first two vertices in the vertex stream. Every vertex after the first two defines a new line, where the other vertex of the line is the previous vertex in the stream. This provides a more dense representation of vertices in situations where the lines to be drawn are connected end to end. Both of these primitive topologies are shown in Figure 3.7.

**Triangle primitives.** The *standard triangle primitive* topologies follow the same paradigm as their line primitive counterparts. The triangle list topology creates a triangle primitive from every three vertices in the vertex stream. The triangle strip creates a triangle primitive out of the first three vertices in the stream, then a new triangle primitive is created for every subsequent vertex, using the prior two vertices in the stream to define the remaining portion of the triangle. As with the line strip primitives, the triangle strip primitive topology can only represent connected "strips" of triangles. These primitive topologies and their ordering are shown in Figure 3.8.

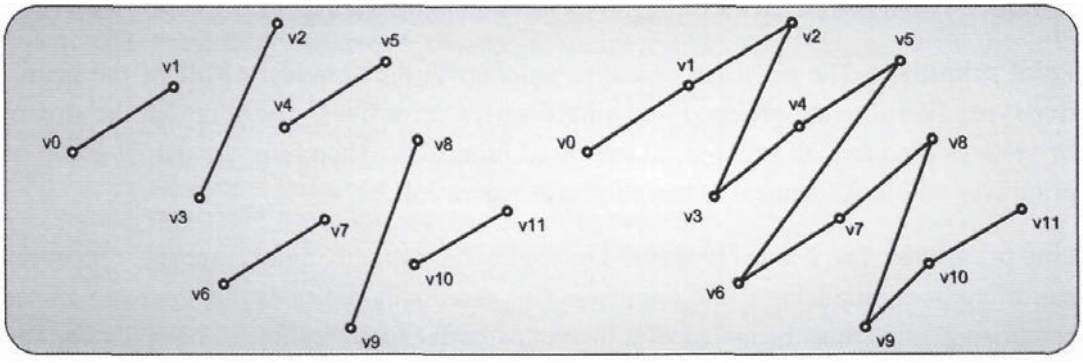



Figure 3.7, Line list and line strip primitives created from our input vertices.

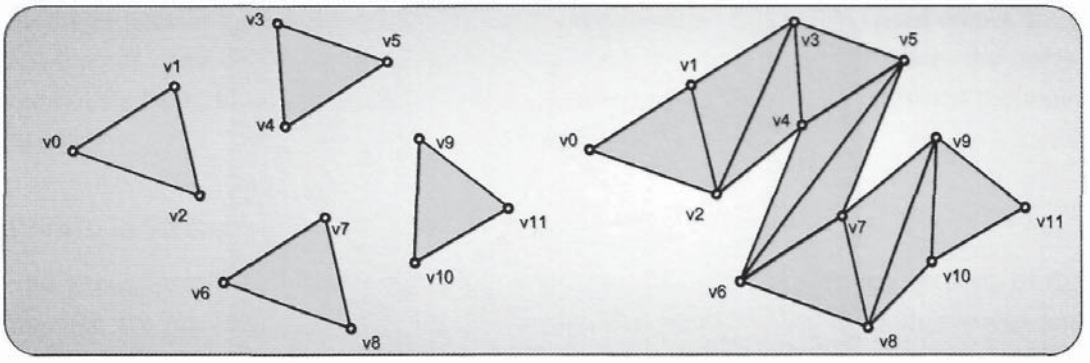



Figure 3.8. Triangle list and triangle strip primitives created from our input vertices.

Primitives **with adjacency.** In some situations, it is desirable to have adjacent primitive information available for use in the pipeline. This means that every primitive will be provided to the pipeline with access to the primitives immediately neighboring it. There are four different primitive topologies that provide adjacency information: *line lists with adjacency, line strips with adjacency, triangle lists with adjacency,* and *triangle strips with adjacency.* These representations provide significantly more information to the pipeline for processing at the primitive level, at the expense of a higher memory and bandwidth cost per primitive. Each of these primitive topologies is shown in Figures 3.9 and 3.10.

Control point primitives. To facilitate the use of higher-order primitives with the tessellation pipeline, additional primitive topology selections are provided. There is one primitive type for each number of control points, ranging from 1 to 32. This provides the ability to perform a very large number of different control patch schemes. One sample control patch primitive is shown in Figure 3.11.

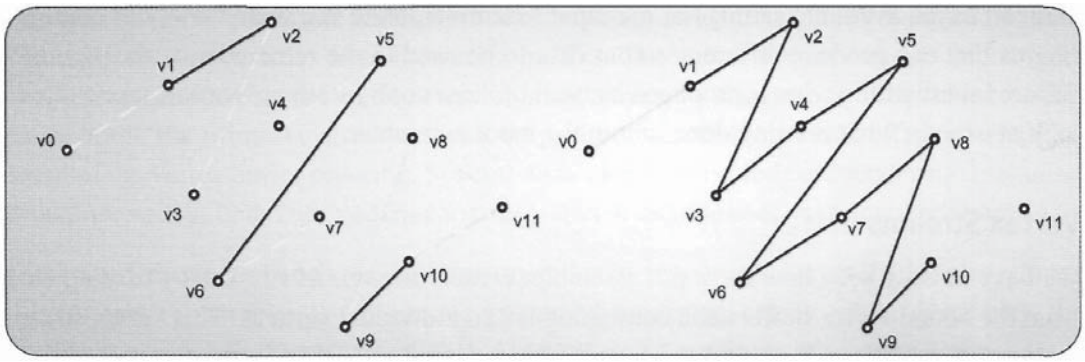



Figure 3.9. Line lists and line strip primitives with adjacency created from our input vertices.

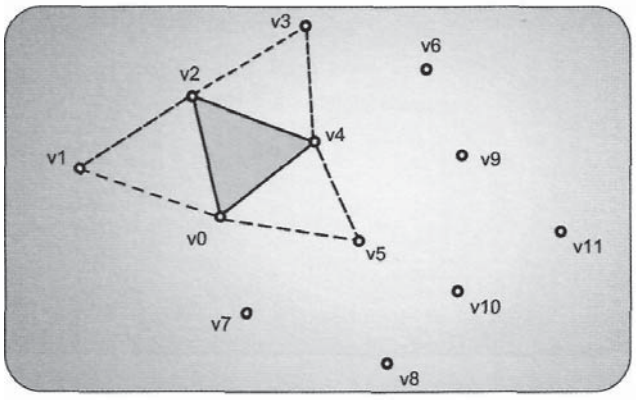



Figure 3.10. Triangle list primitive with adjacency created from our input vertices.

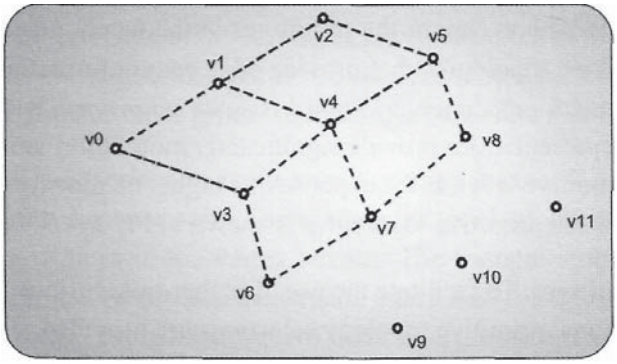



Figure 3.11. Control patch primitives created from our input vertices.

## 3.3.3 Input Assembler Stage Processing

With all of the available settings of the input assembler, there is a wide variety of configurations that can produce different output data to be used in the remainder of the pipeline. Before investigating what is produced by each of these configurations, we will take a closer look at exactly what is being done within the input assembler.

### Vertex Streams

We have already seen how the input assembler creates streams of vertices by reading data from the bound vertex buffers and combining it into individual vertices. This vertex stream produces the vertices in one of two ordered sequences. The vertices are either left in their existing order from their vertex buffers, or they are rearranged into the order specified

by the indices in the index buffer. The determination of which order is used comes from the type of draw call used to invoke pipeline execution. Regardless of where the ordering comes from, the input assembler produces a stream of vertices to represent the input geometry.

### Primitive Streams

The vertex stream is further refined into a stream of primitives. Various portions of the pipeline are intended to operate on only vertices (vertex shader), on both vertices and primitives (hull shader and domain shader), or only on complete primitives (geometry shader). In each of these cases, the stream of primitives is shaped by the selected primitive topology and the type of draw call used to initiate pipeline execution.

### Draw Call Effects

So how exactly are the vertex and primitive streams that result from a draw call influenced by the type of call used? This can best be demonstrated by examining the output from the input assembler for each of the types of draw calls. The following sections provide information about each of the classes of draw calls, and what effects they have on the resulting data streams.

**Standard draw methods.** The simplest draw methods to consider are the Draw() and DrawAuto () methods. Both of these methods trigger the input assembler to assemble vertices based on its input layout settings. This creates a stream of vertices, which is then passed into the pipeline. The primitive information generated by these draw calls is determined by the primitive topology currently specified in the input assembler, and the construction of primitives is dependent on the order of the vertices found in the vertex buffers. This process can be considered the basic vertex and primitive construction process.

**Indexed draw methods.** The first variation on the basic draw calls is the addition of indexed rendering. As we have already discussed, indexed rendering uses the indices of an index buffer to determine which vertices are used to construct primitives instead of simply using the vertex order from the vertex buffers. The vertex stream remains unchanged, with the exception that the vertices within it are chosen by the indices of the index buffer. In addition, the primitive stream creates its primitives with the index buffer ordering, instead of the vertex buffer ordering. Several draw calls support indexed rendering, including DrawIndexed(), DrawIndexedInstanced(), and DrawIndexedInstancedIndirect().

**Instanced draw methods.** In addition to indexed rendering, Direct3D 11 also allows instanced rendering. We discussed instanced rendering briefly while describing the available members of the D3D11_INPUT_ELEMENT_DESC structure. If a vertex component is declared as a per-instance component, and the pipeline is invoked with one of the instanced

draw methods, the vertex and primitive streams are created in largely the same ways that were described for basic rendering and indexed rendering. However, the vertex and primitive streams are repeated for each instance of the object, except that the per-instance vertex components are updated for each complete instance. The number of instances is determined by the parameters passed to the draw call, and the per-instance vertex components are taken from one or more vertex buffers bound to the input assembler.

Overall, the result of using instanced rendering methods is that the vertex and primitive streams are multiplied by the number of instances that are being rendered. There are four different instanced draw calls: DrawInstanced(), DrawIndexedInstanced(), DrawInstancedIndirectQ, and DrawIndexedInstancedIndirect().

Indirect draw methods. In addition to the basic, indexed, and instanced rendering methods, there is another type of draw call: indirect rendering. This technique doesn't actually modify the vertex and primitive streams, but it should be discussed here for a complete view of the available draw methods. Instead, indirect rendering methods allow a buffer resource to be passed as the draw call parameters. The buffer contains the required input information that would normally be passed by the application. The indirect rendering methods are DrawInstancedIndirectQ and DrawIndexedInstancedIndirect(). As an example, the number of vertices, the starting vertex, the number of instances, and the starting instance of the DrawInstancedIndirect() method would be contained within a buffer resource, instead of being directly passed to the function.

The purpose of these calls is to allow the GPU to fill in a buffer, which then can control how a draw sequence is performed. This shifts control of the draw call to the GPU instead of the application and provides the first steps for making the GPU more autonomous in its operations. However, indirect rendering operations don't modify the construction of the vertex and primitive streams—they only modify how the draw method parameters are passed to the runtime.

Mixed rendering draw methods. As we have seen, each class of rendering operation is not mutually exclusive of the others. The names of the draw methods typically include several of these rendering techniques, and provide a variety of different ways to execute the pipeline. In each of these mixed cases, the resulting vertex and primitive streams are a mixture of the independent rendering types.

## 3.3.4 Input Assembler Pipeline Output

### User Defined Attributes

Along with an understanding of what the input assembler can produce, and how to manipulate its output, we need to consider how the output streams interact with the remainder of the pipeline. For the input assembler to produce output that is compatible with the desired

vertex shader, the array of D3D11_INPUT_ELEMENT_DESC structures used to create the input layout object must match the declared input signature of the vertex shader. Therefore, the individual vertex attributes must match what is expected by the vertex shader, including the data type and semantic name. Listing 3.7 shows a sample vertex shader program in HLSL.

```
struct VS_INPUT
{
    float3 position : POSITION;
    float2 tex : TEXCOORDS;
    float3 normal : NORMAL;
};
VS_0UTPUT VSMAIN( in VS_INPUT v )
{

}
```

Listing 3.7. A sample vertex shader input declaration.

For this example, the application would need to provide vertex data that consists of a POSITION semantic attribute with three floating point elements, a TEXCOORDS semantic component with two floating point elements, and a NORMAL semantic component with three floating point elements. Creating an input layout object helps the developer ensure that the vertex data and the vertex shader signature match one another. Since the vertex data is supplied by the application, it can be created or loaded in the proper format to be used with a particular vertex shader. These inputs are then used by the vertex shader to perform its own calculations, and it will then also define a number of output attributes, which it will write out to be used by the next active pipeline stage. This process is repeated for each subsequent pipeline stage, with each stage declaring its output signature and then supplying its output data in them. Some typical examples of input data produced by the input assembler include, but are certainly not limited to, the following:

float 3 position—the position of the input vertex

float3 normal—the normal vector of the input vertex

float 3 tangent—the tangent vector of the input vertex

float 3 bitangent — the bitangent vector of the input vertex

uint4 boneIDs—four-bone IDs of the vertex used for skinning

float4 boneWeights—four-bone weights of the vertex used for skinning

The input assembler can also produce three different system value semantics: SV_VertexID, SV_PrimitiveID, and SV_InstanceID. These system value semantics are not supplied by the user, but are instead generated by the input assembler as it creates the output data streams. The SV_VertexID system value is an unsigned integer, which uniquely identifies each vertex in the assembled vertex stream. It is first available in the vertex shader, and provides a simple way to differentiate between vertices later in the pipeline. The SV_PrimitiveID provides a similar unique identifier for each primitive in the primitive stream. It is first available in the hull shader stage, since the vertex shader doesn't use primitive-level information. Finally, the SV_InstanceID uniquely identifies each instance of the geometry in an instanced draw call. It is first available for use in the vertex shader stage. Together, these three system values provide a fairly extensive method for identifying each of the data elements that the input assembler produces.

## 3.4 Vertex Shader

The first programmable shader stage in the rendering pipeline is the vertex shader. As described above, the programmable shader stages execute a custom function written in HTSL. In the case of the vertex shader stage, the vertex shader program is a function that is invoked once for each vertex in the vertex stream produced by the input assembler. Each of the input vertices is received as the argument to the vertex shader program, and the processed vertices are returned as the result of the function. Each vertex shader invocation is executed in complete isolation from the others, with no communication between them possible. Since it is executed once for each input vertex, there will be a one-to-one mapping of input vertices to output vertices from the vertex shader stage. The location of the vertex shader within the pipeline is highlighted in Figure 3.12.

In addition, each invocation of the vertex shader is not aware of the primitive stream produced by the input assembler. Its sole purpose is to process vertex data and leave any higher-level processing to the stages further down the pipeline. With no knowledge of primitives, the vertex shader operates in the same way for all different topology types. Regardless of whether it is processing geometry that was submitted as points, lines,

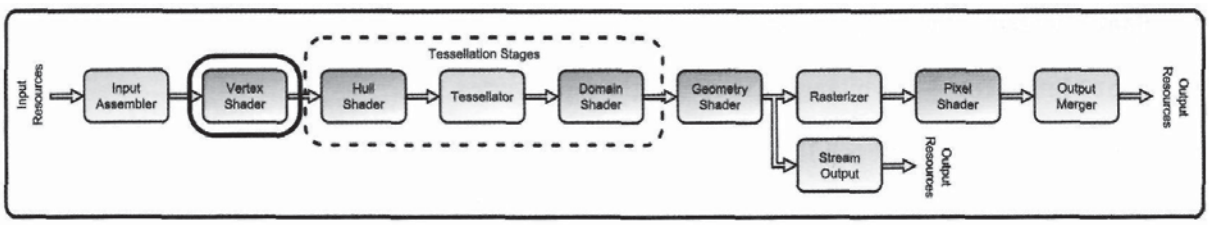



Figure 3.12. The vertex shader stage.

triangles, or patches, the individual vertices are all treated the same by the vertex shader. The processed output from the vertex shader is combined with the primitive information generated by the input assembler stage later in the pipeline. This simplifies the job of the vertex shader stage, allowing it to perform the exact same operations regardless of the primitive type.

The vertex shader is located between the input assembler stage and the hull shader stage. This means that it processes geometry data immediately after it is inserted into the pipeline, and immediately before any tessellation is performed. Some examples of operations that are typically performed in the vertex shader are the application of transformation matrices to input geometry, performing bone-based animation transformations (also known as *vertex skinning*[6]), and performing per-vertex lighting calculations. We will investigate more precisely what the vertex shader stage is typically used for, and what tools it has at its disposal throughout the remainder of this section. We begin by considering what data the vertex shader receives as input followed by the types of processing that it performs, and then finally consider what data it produces as output.

## 3.4.1 Vertex Shader Pipeline Input

Since the vertex shader is located directly after the input assembler in the pipeline, it naturally receives its input from there. All of the work done to configure the input assembler's input layout is intended to make the created vertices match the format expected by the vertex shader stage's current program. This is why compiled shader byte code is required as an input when creating an ID3DllInputl_ayout object—to ensure that the assembled vertices will match what is needed to execute the vertex shader program.

As indicated in the description of the input assembler's output, the vertex shader input can also use system value semantics in addition to the attributes of the assembled vertices. The two system value semantics available as inputs to the vertex shader are SV_VertexID and SV_InstanceID. Both of these provide useful information to the vertex shader program, and the developer only needs to include them in the vertex shader input signature to acquire them. For example, the SV_VertexID can be used as an index into a lookup table for providing pseudo-random values to the shader. Similarly, the SV_InstanceID can also be used as an indexing mechanism. However, it is only updated for each instance generated by the input assembler, so any lookup table values would be applied uniformly to all of the vertices of an instance. This could be used to introduce variation between instances of a mesh to make each instance appear more unique.

We have already described how these system value semantics are available to the vertex shader as inputs. They can also be provided to later stages in the pipeline, but they

[6] Vertex skinning is described in detail in Chapter 8, "Mesh Rendering."

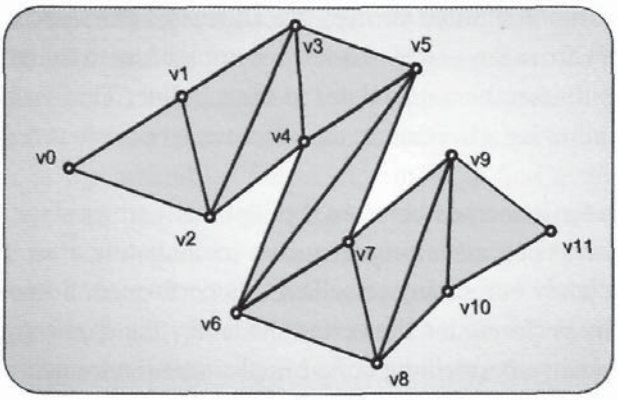



Figure 3.13. A depiction of vertices of a triangle strip that belong to multiple primitives.

must be passed through the vertex shader output if they are to be used later on. This is a reasonable restriction—if the system values are generated in the input assembler, they are directly available to the vertex shader. If that shader decides not to propagate the system values further down the pipeline, they would not simply reappear in a later pipeline stage that needs them. The developer must ensure that any system value semantics that are needed later on are passed from stage to stage until they are used.

One system value semantic that is not available to the vertex shader is the SV_PrimitiveID attribute. SV_PrimitiveID is produced by the input assembler and provides an identifier for each primitive in the primitive stream. Vertices are processed by the vertex shader stage one at a time, regardless of the primitive type being rendered. In addition, a single processed output vertex produced by the vertex shader can be use in more than one primitive, such as in triangle strips, or as seen in our discussion of indexed rendering. This makes having a unique primitive ID for an input vertex attribute illogical, since it can be used in multiple primitives. Figure 3.13 shows a very simple set of geometry that is submitted as a triangle strip, which highlights this situation.

When geometry such as this is processed by the pipeline, each vertex would be processed individually by the vertex shader stage. If the SV_PrimitiveID were allowed in the vertex shader, which primitive index would be assigned to vertex *v37* It belongs to multiple primitives, and hence it would not be clear which primitive ID a shared vertex should use.

## 3.4.2 Vertex Shader State Configuration

As a programmable shader stage, the vertex shader implements the common shader core functionality. This means that it provides a standard set of resource interface methods that allow an application to provide the shader program access to the required resources. As with all pipeline manipulations, all of the methods that can change the vertex shader stage's

state belong to the ID3DllDeviceContext interface. We will review each of these available resources and see how they are used in the context of the vertex shader stage.

### The Shader Program

The first, and probably the most important state of the vertex shader stage is the shader program itself. The process for compiling a shader and creating a shader object from it is described in detail in Chapter 6, so we will assume here that a compiled shader has been used to create a valid vertex shader object. An application can, and normally does, create more than one vertex shader object for use in rendering various objects. To configure the vertex shader stage with the desired shader program, the application must use the ID3DllDevice Context:: VSSetShader method. Listing 3.8 shows an example of how this is performed.

```
// pNextShader must be initialized to the desired shader object.

ID3DllVer-texShader* pNextShader = pShaderl;
m_pCorttext->VSSetShader( pNextShader, 0, 0 );
```

Listing 3.8. Setting a vertex shader program.

The method to set the desired shader object requires three arguments. The first is a pointer to the shader object to bind to the stage. The second and third parameters are used to pass an array of shader class instances to be used in the shader program. These class instances are objects that can be used to provide a portion of a shader program with an external object, and are intended to reduce the number of individual shader objects needed to support a wide variety of similar rendering scenarios.[7] There is also a corresponding ID3DllDeviceContext : :VSGetShader method, which can be used to retrieve the shader and class instance objects that are currently bound to the vertex shader stage. This can be useful to read an existing shader object setting, so that it can be restored after an interim operation is performed.

### Constant Buffers

The next configuration that the application can provide to the vertex shader program is an array of constant buffers. Constant buffers are described in detail in Chapter 2. These are essentially buffer resources that the application loads with parameters that are made available to the shader program, which can use the data directly within the HLSL code. This is used to communicate parameters to the shader program that will remain the same throughout a pipeline execution—hence the name *constant* buffers. The process for filling a constant buffer with data is also described in detail in Chapter 2, so we will focus here on

[7] These class instances are described briefly in Chapter 6, but are not discussed in great detail. The reader is referred to the DXSDK documentation for further guidance on how to use these objects.

how to bind these resources to the vertex shader stage. Listing 3.9 shows how to bind one or more constant buffers to the vertex shader stage.

```
ID3DllBuffer* cbuffePS[D3Dll_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT];

// Fill each element of the array with buffers here___

pContext->VSSetConstantBuffers( 0, count, cbuffers );
```

Listing 3.9. Binding an array of constant buffers to the vertex shader stage.

Listing 3.9 shows how to set multiple constant buffers with a single device context method. In fact, this example shows how to set all of the available constant buffer slots. This can be an advantageous technique if all of the elements of the ID3D11Buffer* array are initialized to NULL. Then any constant buffers that were previously attached to this stage are automatically detached when the NULL value is bound in its place. Figure 3.14 depicts the constant buffers slots in the vertex shader before and after such a "complete" state setting. The array of constant buffer pointers that are currently set in the vertex shader stage can also be retrieved with a corresponding ID3D11DeviceContext:: VSGetConstant

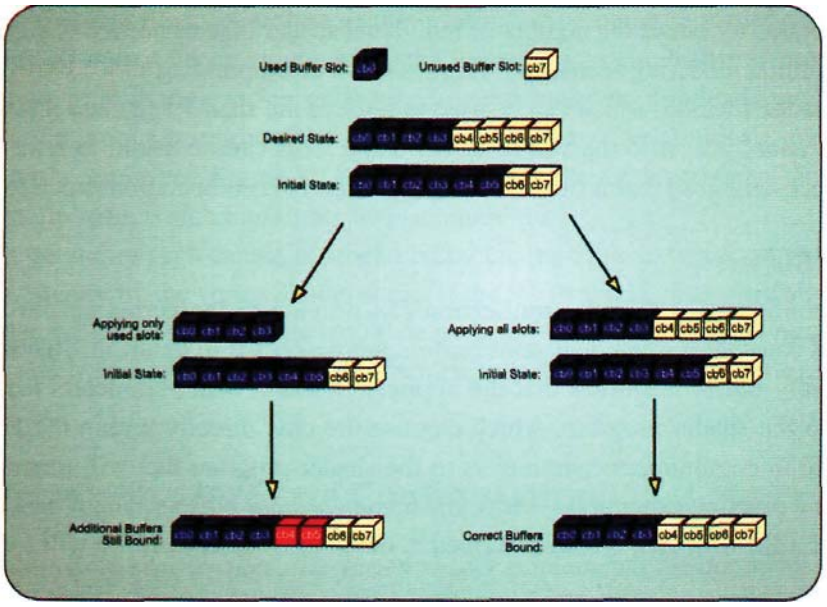



Figure 3.14. Unbinding unused constant buffers by using an array of buffer resource pointers initialized to NULL.

Buffers method. Once again, this can be useful for reinstating an existing setting after an interim operation is performed.

### Shader Resource Views

For the vertex shader program to be able to access the various types of read-only resources that are available to it, the resources must be bound to the vertex shader stage with a shader resource view. Both buffer resources and texture resources are bound to the pipeline in exactly the same manner, using the ID3D11DeviceContext: :VSSetShaderResources method. As with constant buffers, one or more shader resource views can be set simultaneously with the same method call. Listing 3.10 demonstrates how this method is used.

```
ID3DllShaderResourceView*
ShaderResourceViews[D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT];

// Fill the array with the desired resource views...
pContext->VSSetShaderResources( start, count, ShaderResourceViews );
```

Listing 3.10. Binding an array of shader resource views to the vertex shader stage.

The array of shader resource views operates in much the same fashion as described for the constant buffers. Once a shader resource view has been set in one of the slots, it will remain bound in this location until it is replaced by either another shader resource view or a NULL pointer. The technique of clearing out all unused shader resource views with NULL in between pipeline executions takes on more importance when applied to resources. If a resource that is dynamically generated by the GPU is used as a shader resource in another rendering pass, it is first written to by the pipeline in the first pass, and then read in the shader program in the second pass. Writing to the resource is done with either a render target view or an unordered access view, while reading is performed with a shader resource view. If a resource is bound for reading with a shader resource view and is mistakenly left bound to the vertex shader stage, any attempt to bind the resource for writing will generate an error, since a resource cannot be simultaneously read and written by multiple views. This situation is depicted in Figure 3.15. The simplest way to ensure that this does not happen is to clear all unneeded shader resource views from the vertex shader stage each time the pipeline is configured for a new rendering effect.

### Samplers

The final configuration that the application can supply to the vertex shader stage is *sampler objects*. Samplers provide the ability to perform various types of filtering on textures

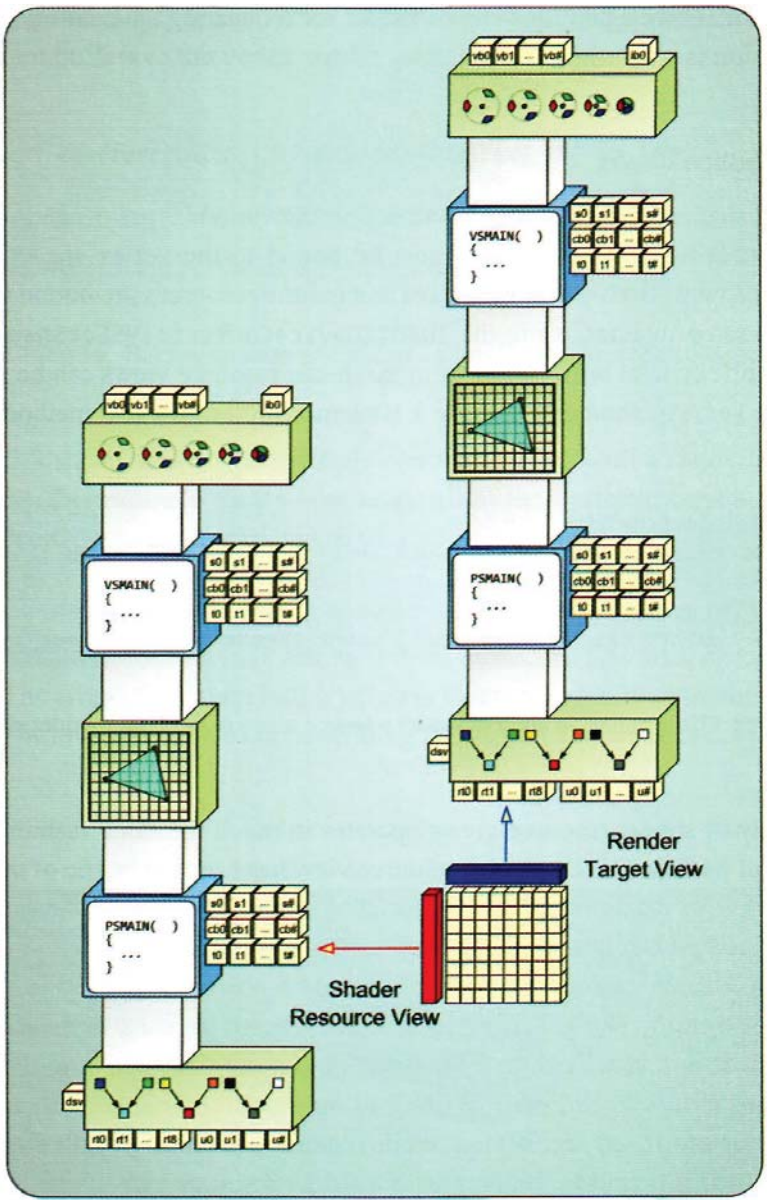



Figure 3.15. A resource being used in two different rendering passes for reading and writing.

when they are read from memory.[8] This often leads to significant performance and/or quality increases for the sampled values, since the GPU typically has additional specialized hardware specifically intended to accelerate this process. Listing 3.11 demonstrates how ID3DllSamplerState instances are bound to the pipeline. Once again, we see the familiar

[8] Samplers were described in detail in Chapter 2.

method of simultaneously setting an array of sampler states to reduce the number of interactions with the API required to configure the pipeline for rendering.

```
ID3DllSamplerState*    SamplerStates[D3D11_C0MMONSHADER_SAMPLER_SLOT_C0UNT];

// Fill in the sampler states here...
pContext->VSSetSamplers( start, count., SamplerStates );
```

Listing 3.11. Binding an array of sampler state objects to the vertex shader stage.

Overall, we can group these four configurations together and consider them to be the available vertex shader stage states. Configuring the vertex shader requires the proper shader program, along with all the constant buffers, shader resources, and samplers that are called for in the shader program. Since each vertex shader program can have significantly different uses and requirements, it is possible to have many different combinations of these states.

## 3.4.3 Vertex Shader Stage Processing

We now know what data the vertex shader can receive as input data from the input assembler, and which resources can be bound by the host application. We also know that the vertex shader program provides custom processing of individual vertices, regardless of which topology the pipeline has been invoked with. So, what types of operations are performed in the vertex shader program? Are certain types of operations better suited to this stage's semantics? We will consider what operations the vertex shader stage is typically used for and will comment on how best to use this stage of the pipeline.

### Geometric Manipulation

The vertex shader is traditionally used to perform geometric manipulations on input data before it is rasterized later on in the pipeline. This is because the vertices of a model hold the information that represents the geometric surface of the object being rendered. Since all vertices are passed through the vertex shader early in the pipeline, this is a logical place to perform geometric manipulations. A few examples can clarify this topic fairly easily.

The prototypical operation normally performed in the vertex shader is to apply transformation matrices to vertex positions. This operation modifies the input position attribute of the vertices to reflect the placement of the geometry within the scene by performing a matrix multiplication of the vertex position with a transformation matrix supplied by the

application in a constant buffer.[9] In addition, any vertex attributes that are sensitive to the model's coordinate system must also be transformed into the new coordinate system with a similar matrix multiplication. Both the position of the vertices and their normal vector define the physical representation of the model, which is modified by the transformation matrices. By performing this calculation in the vertex shader, we can be certain that all of the model's geometry will be converted to the desired coordinate space before further processing is performed further down the pipeline.

Another operation frequently performed in the vertex shader stage is *vertex skinning.*[10] This performs a different type of coordinate transformation on a model, in which bones are hierarchically linked to one another within a system of bones, and each vertex is assigned to one or more bones. These bones move different parts of the model in different ways, for example, to simulate how your bones move your skin when you move your arm. Like the standard transformations example above, this operation modifies the physical structure of the model to provide the desired pose. It is also appropriate to perform this operation in the vertex shader, before further processing.

## Vertex Lighting

Geometric manipulation and vertex skinning can be efficiently performed in the vertex shader, since they can be performed directly on the data that the vertices represent—the physical shape, structure, orientation, and location of the geometry being rendered. Of course, these are not the only types of operations that can be performed in a vertex shader. Because this stage is programmable, many special types of processing can be performed in the vertex shader and then passed to later stages. A common example of this type of operation is *per-vertex lighting.* Per-vertex lighting calculates the amount of light reflected from a vertex, normally using some form of simplified lighting model (Hoxley). The lighting data can be stored as a three- or four- component floating point attribute in the output vertex format, where a value of 1.0 means that a full amount of the color is available and a 0.0 represents the complete absence of that color.

These per-vertex lighting values are passed down the pipeline and eventually arrive at the rasterizer stage, which interpolates them as attributes between each vertex in the primitive. These interpolated values are then applied to each pixel generated by the rasterizer, and are eventually used to determine the final color written to the render target. By performing these calculations in the vertex shader, lighting quantities are calculated only at a sparse set of points (at the vertices), instead of at every pixel that a model generates. The generated values are interpolated between the vertices, making the reduced resolution of the calculation less noticeable, as long as the vertices are not too far apart on the screen.

[9] A basic introduction to transformation matrices, as well as coordinate spaces, is provided in Chapter 8. "Mesh Rendering."

[10] Vertex skinning is also discussed in detail in Chapter 8, "Mesh Rendering."

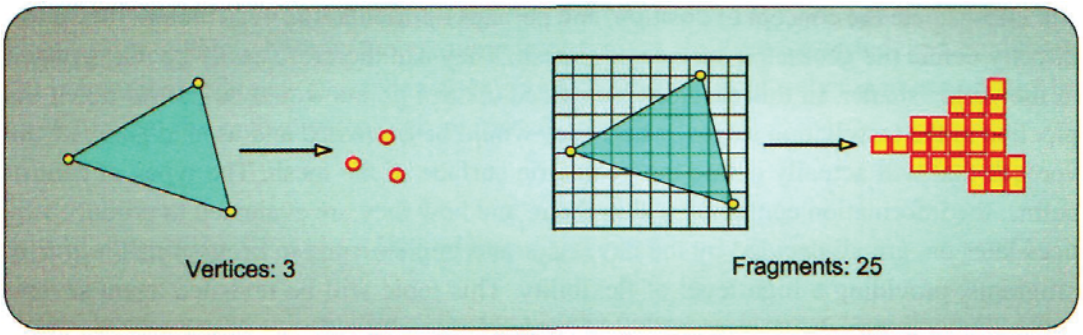



Figure 3.16. The difference between per-vertex and per-pixel calculations in a pair of triangles.

### Generic Per-Vertex Calculations

Many other types of calculations can follow a model similar to the one described above for lighting. If a calculation can be performed in the vertex shader stage and an interpolated version of the data can be used at the per-pixel level, then it makes sense to perform the calculations in the vertex shader where they will only be performed once per vertex. If the calculations were instead performed after rasterization, they would be performed many more times. Figure 3.16 shows the difference in calculation frequency between a per-vertex calculation and a per-pixel calculation.

Mathematically speaking, if the calculation is a linear combination of its inputs, then the results of the per-vertex calculation will be identical to those of the per-pixel calculations, because of the interpolation between vertex attributes. Even if a calculation is not linear, per-vertex calculations can still provide a fairly good approximation of the full per-pixel calculations. The effectiveness of this approximation depends on the type of calculation, as well as on how large the final rasterized primitive will appear in the final render target. If the number of pixels generated between vertices is very small, any inaccuracy of the interpolated values is less likely to be noticed. This also means that if an input set of geometry is created to have a larger number of vertices, the interpolation effects will be less noticeable, at the expense of an increased number of vertices to process. There is no simple rule to determine the required level of detail for a particular model. This is a function of the available processing power and the desired image quality. As a small indication of what is coming later in this chapter, the new tessellation stages are aimed at striking a balance in this regard, by dynamically generating an appropriate number of vertices, only where they are needed (where they are visible). This reduces the overall vertex processing costs, while still producing small screen-space-sized primitives, which effectively raises the available image quality for the same amount of required processing.

### Control Point Processing

One other way to use the vertex shader stage is to process data that is actually not vertices, but rather control points of a higher-order primitive or control patch. These control points

still encapsulate the concept of position, and perhaps of orientation, even though they don't directly define the geometric surface of a mesh. They can therefore easily be manipulated in the vertex shader. In this case, the processed control points would be passed down the pipeline to the tessellation stages, where they would be evaluated and used to generate the vertices that will actually define the geometric surface of the mesh. The types of control points, the information contained within them, and how they are evaluated to produce vertices later on, are all decided by the developer and implemented in programmable shader programs, providing a high level of flexibility. This topic will be revisited again several times throughout the book (especially in Chapter 4, "The Tessellation Pipeline," as well as in several of the sample algorithm chapters).

### Vertex Caching

Earlier in this chapter, we discussed how the vertex shader program is invoked once for each vertex. While the vertex shader stage is effectively executed once per vertex, individual processed vertex results may be shared among more than one primitive. For example, if a vertex is shared by two triangle primitives in a triangle strip, it can be processed once, and its result can be used in both triangle primitives later in the pipeline, after the vertex shader stage. Using any of the "strip" primitive topology types will allow this type of vertex reuse. In addition, when using indexed rendering even the "list" primitive topology types can define multiple primitives which share the same vertices. This can significantly lower the number of vertices to be processed when compared to rendering techniques that cannot share vertices. As an example, consider the geometry shown in Figure 3.17. In each of these three cases, different primitive topologies are used to define the required model, with varying numbers of vertices to be processed.

One additional consideration regarding vertex caching is that this operation is hardware dependent. The size of the cache (and even its very existence) can vary significantly among different CPUs. Therefore, it is a best practice to ensure that any shared vertices are always referenced as closely together as possible in the vertex or input. In the case of the

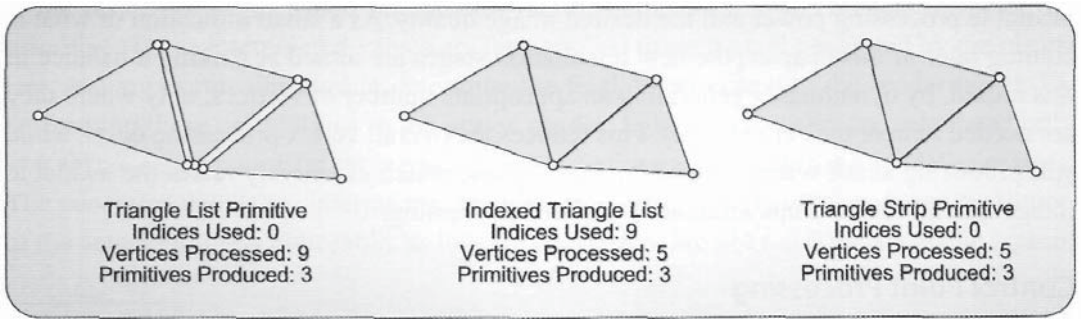



Figure 3.17. Varying sets of geometry with different topologies, and the resulting number of required vertices to be nrocessed.

"strip" primitive topology types, this is automatically done, since each primitive uses the vertices immediately before it to create new primitives. In the case of indexed rendering, the indices that reference the same vertex should be located as closely together as possible, to ensure that the cached vertex can be reused.

### 3.4.4 Vertex Shader Pipeline Output

When deciding what information to include in the output vertex structure, there are some considerations to take into account regarding how the remainder of the pipeline will be used. Figure 3.18 shows a block diagram of the rendering pipeline.

The vertex shader stage is followed by the group of tessellation stages (the hull shader, tessellator, and domain shader stages), then the geometry shader, and then the rasterizer stage. Depending on which of these stages are active between the vertex shader and rasterizer stages, different requirements must be met. If the vertex shader is connected directly to the rasterizer stage, it must produce the final clip space position of each of the vertices in the SV_Position system value semantic. If the tessellation stages (all three are either active or inactive together) or the geometry shader stage is active, then it is optional for the vertex shader to produce the clip space position. However, the last active stage immediately before the rasterizer stage must provide the SV_Position system value semantic.

Similarly, if the tessellation stages are active, the vertex shader stage must provide control points to the hull shader stage. The distinction between an output vertex and a control point really depends only on the tessellation scheme being implemented, but in both cases the data is still produced by the vertex shader. If the tessellation stages are disabled and the geometry shader is enabled, the vertex shader output is sent directly to the geometry shader. In this scenario, the geometry shader must supply the SV_Position semantic to the rasterizer, although it could be calculated in the vertex shader and then passed to the geometry shader. The wide variety of options for producing just one position calculation underscores the flexibility that the pipeline provides, and the corresponding freedom for the developer to implement an algorithm in the most advantageous method available.

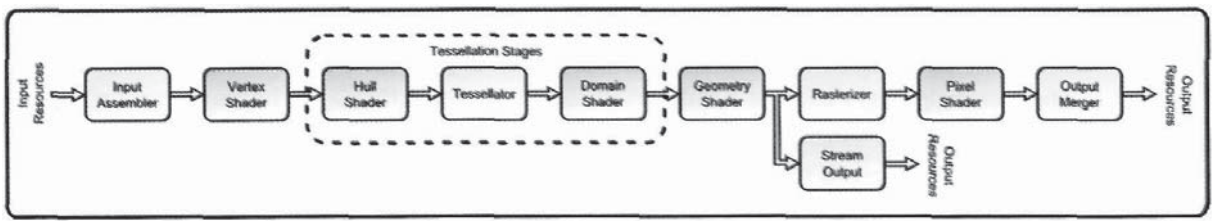



Figure 3.18. A block diagram of the rendering pipeline.

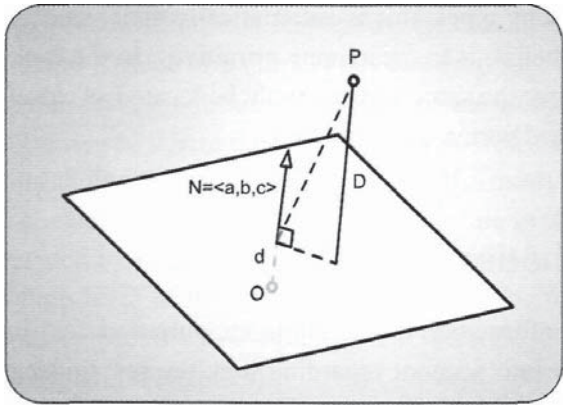



Figure 3.19. Calculating the distance from a point to a plane.

## System Values

Besides these stage-based considerations, two new system values are available for the vertex shader stage to write to: SV_ClipDistance[n] and SV_CullDistance[n]. Both are available for writing in the stages before the rasterizer, where they are used to perform two different types of operations: *clipping* and *culling*. Clipping and culling for the pipeline are implemented in the rasterizer, and are discussed in more detail in the "Rasterizer" section of this chapter. However, we will describe these operations briefly here to explain the functionality provided in these two system values.

We begin with a brief mathematical introduction to point-to-plane distance calculations. In general, the shortest distance from a point to a plane can be found by taking the dot product of the point's position and the normalized normal vector of the plane, minus the shortest distance from the plane to the origin of its coordinate space. This result is easily obtained when the familiar plane equation is available for a plane. Equation (3.1) shows the equation that provides this property, where *a, b,* and *c* are the components of the normal-length normal vector, and *d* is the shortest distance from the origin to the plane. This equation produces a scalar result, which can take three different value ranges. It can be a positive value if the point is on the side of the plane that the normal points into, a zero value if the point is exactly on the plane, or a negative value if the point is on the side of the plane pointing away from the normal vector. Figure 3.19 shows this calculation for a plane and a point:

$$D = ax_1 + by_1 + cz_1 - d. \tag{3.1}$$

To conservatively cull a complete primitive from further processing, all of the vertex positions must be completely outside of at least one of the six planes defining clip space." Each of the distances between the vertex positions and each of the planes can be calculated as described above. If all of the vertices of a primitive result in negative distance values for any of the six planes, the primitive can be discarded since it would not be visible. The SV_CullDistance[n] system value semantics behave in much the same way as the form of conservative culling mentioned above. Each attribute declared with this system value represents the distance to a culling plane. When these system values are interpreted by the rasterizer stage, it will eliminate any primitive whose vertices all have a negative value in the same register. An example of this type of system value usage is shown in Listing 3.12. If the same component of the "clips" attribute were negative for all of the vertices of a primitive, it would be culled without being rasterized at all. This arrangement allows for the results of up to four different culling equation results to be stored for this attribute.

```
struct VS_OUT
{
    float4 position : SVPosition;
    float4 clips    : SV_CullDistance;
};
```

Listing 3.12. A sample output vertex structure that uses a four-component SVCullDistance system value semantic.

The SV_ClipDistance system value semantic operates in a similar fashion, and defines the distance to a clipping plane. If one or more vertices of a primitive have negative values in this attribute, its value will be used to determine which portion of the current primitive should be clipped. The actual clipping can be performed before or after rasterization—this is an implementation-specific detail that the developer can't know in advance. Regardless of how the primitive is clipped, the result is that none of the fragments generated after rasterization will contain a negative value in the interpolated SV_ClipDistance system value semantic. This differs from the SV_CullDistance system variable, which performs the culling test on the entire primitive instead of trying to "clip" the primitive to only have a positive SV_ClipDistance attribute.

The maximum number of these clip and cull system values is a total of two attributes, with up to four components, each of which can be declared in any combination of SV_ClipDistance and SV_CullDistance. This means that a total of eight planes can be either clipped against (if two float4s are used as SV_ClipDistance) or culled against (if two float4s are used as SV_CullDistance) or some combination of the two.

[11 1] We discuss clip space and its properties in more detail in the rasterizer and pixel shader sections of this chapter.

## 3.5 Hull Shader

Our next stop in the pipeline brings us to the new tessellation stages. These three stages work together to implement a flexible and configurable system. The stages, in order of appearance in the pipeline, are the hull shader stage, the tessellator stage, and the domain shader stage. Of these stages, the hull shader and domain shader stages are programmable, while the tessellator stage performs fixed-function operations. The hull shader stage can be thought of as performing the setup operations that prepare the input primitives for further processing. It instructs the Tessellator stage how finely to split up the input geometry, and provides the processed control point primitives[12] for the domain shader stage, which will fill in the freshly tessellated vertices with data. Thus, the hull shader plays a very important role in configuring the overall tessellation scheme. The location of the hull shader within the pipeline is highlighted in Figure 3.20.

The responsibilities of the hull shader stage are divided into two separate HLSL functions. The first is the hull shader program. The hull shader program is invoked once for each control point needed to create an output control patch. It receives input control *patch* primitives constructed from the control points produced by the vertex shader, and must create one processed output control *point* for each invocation of the hull shader program. Unlike the vertex shader, the hull shader program is aware of both the input and output primitive topology of the geometry being passed through it. In fact, *every* invocation of the hull shader has access to all of the input control points that belong to its input control patch. With access to the complete input data, the hull shader stage can also statically expand or reduce the number of control points in the control patch before the control patch is passed to the next stage in the pipeline. This change in control point count is statically declared with a special attribute in HLSL, and all control patches passed in to the hull shader will be reduced or expanded in the same way. This is the first time we have seen a pipeline stage that can perform an amplification of the primitive data stream passing through it.

The second HLSL function that performs operations within the hull shader stage is referred to as a patch constant function. In contrast to the hull shader program, the patch

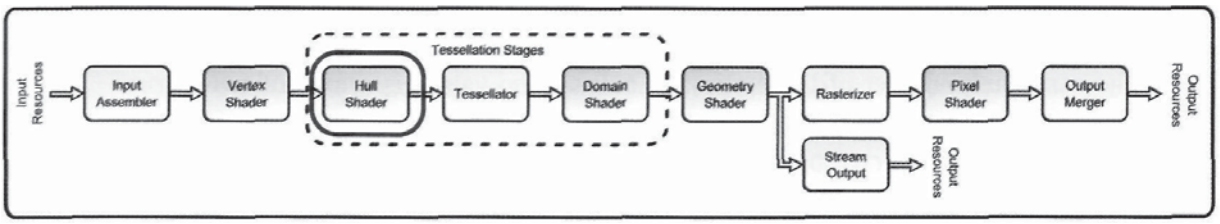



Figure 3.20. The hull shader stage.

[12] The control patch primitive types are described in the "Primitive Topology" section earlier in this chapter.

constant function is only executed once for each complete control patch. This function's primary purpose is to define several tessellation factors, which tell the tessellator stage how finely to tessellate the input primitives. It can also calculate additional attributes that are constant for a complete control point patch, which are then supplied to the domain shader later in the pipeline. The hull shader stage is the only programmable pipeline stage that is required to provide two functions in HLSL.

Throughout the remainder of this chapter, we will further explore how this combination of functions allows the hull shader stage to enable a wide variety of tessellation algorithms. We begin by examining the typical inputs to the stage, and what types of states are available for configuration. This is followed by a discussion of typical processing operations that are suitable for the hull shader. Finally, we will look at the data produced by the hull shader stage, and where in the pipeline it is passed on to.

## 3.5.1 Hull Shader Pipeline Input

The hull shader stage is situated immediately after the vertex shader stage in the pipeline. Its input data is a set of processed vertices produced by the vertex shader, which is combined with the primitive stream information produced by the input assembler stage to form control patch primitives. Strictly speaking, the hull shader program operates on complete control patches, which are made up of control points, and not vertices. However, there is little distinction between control points and vertices. Both vertices and control points define a position, along with some additional attributes, so there is not much difference in their contents. The only difference between control points and vertices is actually determined by the primitive topology specified in the input assembler stage. If the pipeline configuration for a particular draw call will use the tessellation stages,[13] it must declare one of the control point patch list primitive types. If an equivalent topology is passed instead (a triangle list topology in the place of a three-point control patch list, for example) the runtime will issue an error. Because of this, we can assume that if the tessellation stages are active, we can say that the vertex shader processes control points. If the tessellation stages are not active, we say that the vertex shader as processes vertices.

As mentioned above, the hull shader program is executed once for each of the desired output patch's control points. Every invocation of the hull shader program has full knowledge of all of the control points in the current input control patch. These points are received as a pipeline input attribute, which is declared in a template-like fashion. Two template parameters are declared in the attribute: the structure of the data produced by the vertex shader, and the number of points in the input control patch. The number of points in the control patch must match the number specified in the input assembler's topology type.

[13] The geometry shader stage is also capable of receiving control patches, as well as the tessellation stages.

Listing 3.13 shows a sample attribute declaration for a hull shader program for a three-point control patch list topology.

```
HS_CONTROL_POINT_OUTPUT HSMAIN( InputPatch<HS_CONTROL_POINT_INPUT, 3> ip,
                                uint i : SV_OutputControlPointID,
                                uint PatchID : SV_PrimitiveID )
{
    HS_CONTROL_POINT_OUTPUT output;

    // Insert code to compute Output here,
    output.WorldPosition = ip[i].WorldPosition;

    return output;
}
```

Listing 3.13. A sample hull shader program, demonstrating its attribute declaration.

Listing 3.13 also shows a pair of system value attributes. The particular control point that a hull shader invocation is processing is indicated in the SV_OutputControlPointID system value, which must be declared as an input attribute. This system value semantic provides a single-component unsigned integer that identifies the current hull shader invocation control point within the specified output control patch size. By using this attribute, the hull shader program can determine which portion of the input control points to use when generating the output control point. The second system value is the SV_PrimitiveID, which provides a unique single-component unsigned integer identifier for each control patch. This is the first pipeline stage where this system value semantic is available, since the vertex shader has no concept of primitives.

The patch constant function runs only once for a complete control patch primitive that is passed into the hull shader stage. Like the hull shader program, it has access to the complete set of input control points, and all their attributes. It can also use the SV_PrimitiveID system value, but not the SV_OutputControlPointID system value semantic. This makes sense, because this function operates once for the entire patch—it would have no use for an identifier of individual control points.

## 3.5.2 Hull Shader State Configuration

Since the hull shader stage is one of the programmable shader stages and has access to the same set of resources that have been discussed for the vertex shader. They are covered here for completeness, but in fact, the names of the methods used to set and get the resources are simply updated to indicate which pipeline stage should be affected by the method, so we won't repeat the code listings here. The method names are provided here for reference.

```
ID3DllDeviceContext::HSSetShader()

ID3DllDeviceContext::HSSetConstantBuffers()

ID3DllDeviceContext::HSSetShaderResources()

ID3DllDeviceContext::HSSetSamplers()
```

In addition to the resources that can be bound to the hull shader stage by the application, a number of other function attributes can be specified in the HLSL code. These are also described below.

### Shader Program

The hull shader program is compiled in the same way as any of the other HLSL programs.[14] The distinction is that the hull shader program must be preceded by the patch constant function, and immediately preceded by a list of function-level attributes. These items must all be located within the same HLSL file and compiled into shader byte code, along with the main hull shader program. The resulting byte code is then used to create an instance of the ID3DllHullShader object, which is then bound to the pipeline through the device context.

### Constant Buffers

The primary means of communicating data to the hull shader program is the constant buffer mechanism. This operates in precisely the same way shown for the vertex shader stage. The same the recommendations regarding the persistence of setting of constant buffers between pipeline executions applies here.

### Shader Resource Views

Another possible source of read-only data for the hull shader program is the shader resource views. These provide read-only access to a wide array of resources, as discussed in Chapter 2. Resources provide a larger pool of available memory to the hull shader than constant buffers do, but they can also potentially be slower to access.

### Samplers

Once again, samplers provide a mechanism for performing filtering operations on texture resources. The hull shader stage has access to this functionality, just as discussed for the vertex shader.

[14] See Chapter 6, "The High Level Shading Language," for details about how to compile and create shader objects.

### Function Attributes

To control the tessellation scheme used in the three tessellation stages, several *function attributes* are required in the HLSL code of the hull shader. These attributes are different than the input and output pipeline data flow attributes passed between pipeline stages. Instead, these are individual statements that perform a specific configuration of the tessellation system. The attributes must be located above the hull shader program in the HLSL source code, since they are used to validate the shader program itself. The list of attributes is provided in Listing 3.14. Each attribute will be described when its subject area is discussed throughout this section. We list them here, followed by a brief description.

```
// Specifies the 'domain' of the primitive being tessellated.
[domain("tri")]

// Specifies the method to tessellate with.
[partitioning("fractional_even")]

// Specifies the type of primitives to be created from the tessellation.
[outputtopology("triangle_cw")]

// Specifies the number of points created by the Hull Shader program (which
// is also the number of times the hull shader program will execute).
[outputcontrolpoints(3)]

// Specifies the name of the patch constant function.
[patchconstantfunc("PassThroughConstantHS")]

// Specifies the largest tessellation factor that the patch constant
// function will be able to produce.  This is optional and provided as a
// hint to the driver.
[maxtessfactor(5)]
```

Listing 3.14. Function attributes of the hull shader program.

## 3.5.3 Hull Shader Stage Processing

The processing tasks intended for the hull shader stage are split between the two HLSL functions that an application must provide. Since the hull shader program creates the output control points that are passed down the pipeline, its primary responsibility is to read the input control patches that were produced earlier in the pipeline, and translate them to the desired output control patch format. The patch constant function is responsible for determining the necessary tessellation factors according to a developer-defined metric. In the following sections, we will examine some higher-level concepts that these functions may be used to perform.

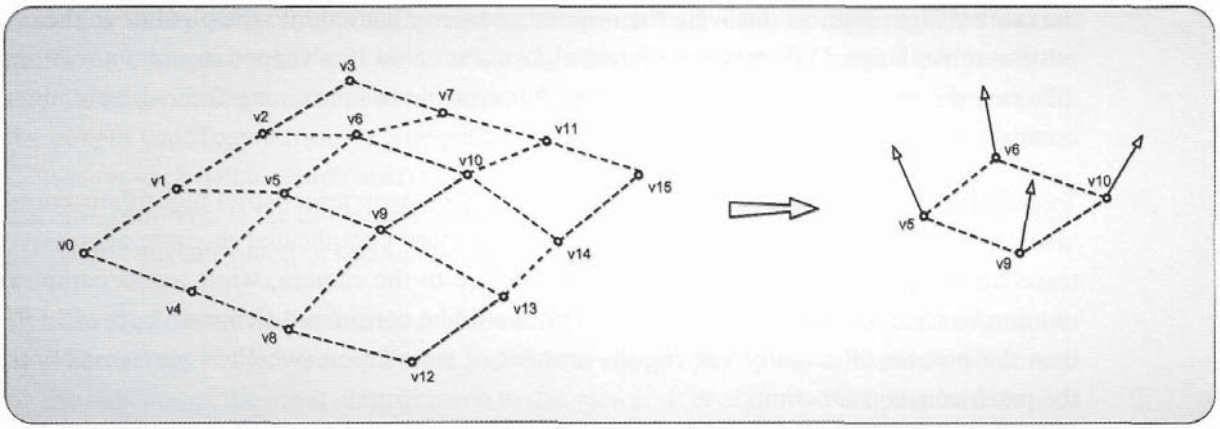



Figure 3.21. Changing the number of control points in the control patches passed on to the domain shader stage.

## Hull Shader Program

As mentioned above, the hull shader operates once for each output control point that needs to be produced for the selected control patch output size. The number of output control points is configured by the application by specifying it in the `outputcontrolpoints` function attribute. The system value semantic input attribute SV_OutputControlPoint identifies which output control point is being created in each invocation of the hull shader program. Its values range from 0 to *n*-\, where n is the number of output control points defined in the `outputcontrolpoints` function attribute. This function attribute is the mechanism by which the hull shader can expand or reduce the number of points in the output control patch. To produce these output control points, the hull shader program receives the complete set of control points that are included in the input control patch primitive as an input attribute. This concept is visualized in Figure 3.21 for a hull shader that receives a 16-point control patch and produces a four-point control patch.

Tessellation **algorithm setup.** This output control patch is the data that will later be used in the domain shader stage. Because of this configuration, the hull shader can be thought of as somewhat of a data setup stage for the tessellation system. As we just saw, the format of the input control patches can be modified, expanded, or reduced in the hull shader. This functionality can be used to produce a control patch type that is compatible with the desired tessellation algorithm implemented in the domain shader stage. With this in mind, the hull shader could be used to consume a standard control patch primitive type (perhaps mandated by the content creation tool chosen for a given project) and then produce the required output patch format for the desired type of tessellation algorithm. This acts as an isolation mechanism for the tessellation system, since a standard input can be provided, and the output from the hull shader stage will always appear in the appropriate format for

the current algorithm, as if it were the original geometry passed into the pipeline in the input assembler stage. Different tessellation algorithms could be swapped in and out without affecting the rest of the pipeline, as long as it is capable of consuming the available input control patch geometry.

**Tessellation algorithm level of detail.** This ability to swap tessellation algorithms could also be used as a form of level-of-detail, where a more complicated or more aggressive tessellation scheme can be chosen for objects close to the camera, while a less complex one can be used for objects farther away. This would be considered a coarser form of LOD than the process of actually varying the amount of tessellation (which is performed with the patch constant function).

**Generic control point calculations.** The hull shader program can also be used for a variety of other processing tasks not related to tessellation. Since it is the last stage before tessellation actually occurs, it is a good candidate for performing calculations that will be shared by more than one of the tessellated points. This can effectively pull the calculations back to an area of the pipeline before the data amplification of the tessellation procedure, saving significant processing costs.

### Tessellation Factor Calculations

The idea behind the patch constant function is that it will calculate two tessellation factors, which are written to special system value semantics. These two system values semantics tell the fixed function tessellator stage how finely to tessellate the primitive being generated from the input control patch. The number of tessellation factors required is influenced by the domain function attribute mentioned above. The type of primitive that is conceptually being split up by the tessellator stage is specified in this attribute. Depending on which type of domain is specified (isoline, triangle, or quad) the number of required edge tessellation factors will vary.

The patch constant function itself is declared with another attribute—`patchconstantfunc`. This simply identifies which HLSL function should be executed for the patch constant function. The remaining function attributes listed above are used to configure execution of the tessellation stage. The `partitioning` attribute configures the type of tessellation to perform, the `outputtopology` attribute defines the type of output primitives that will be created by the tessellation, and the optional `maxtessfactor` attribute tells the driver the maximum amount of tessellation it should expect, so that it can allocate an appropriate amount of memory to hold the results.

These attributes and their available values will be discussed in detail in Chapter 4, "The Tessellation Pipeline." However, the important concept to take away from these configurations is that the patch constant function and the function attributes specified with it are used to determine precisely what the tessellator stage produces. In many ways this

is similar to the setup functionality discussed for the hull shader program. The function attributes configure what will be tessellated (with the domain attribute), how it will be tessellated (with the `partitioning` attribute), and what format its output will be in (with the `outputtopology` attribute). The patch constant function then produces the tessellation factors for each patch, which instruct the tessellator stage how finely to chop up the chosen domain primitive. If the hull shader program sets up the data of the tessellation algorithm, the patch constant function sets up the actual tessellation machinery.

**Fine level of detail.** The calculation of the tessellation factors can be based on any number of different criteria. For example, it is possible to modify the amount of tessellation based on the distance of the input control patch to the viewer. It is of course better to have higher tessellation closer to the viewer than when a patch is farther away, to reduce the number of required computations. However, in other cases this approach is either not appropriate or not ideal. For example, when an object is relatively smooth, with only a few sharp details, it may be better to base the tessellation factors on some measure of the amount of change in a surface. This could either be precalculated and stored in the control points, or perhaps stored in a texture for lookup later on. It is also possible (and in fact, more probable) to combine more than one heuristic approach for this problem. Examples of this type of analysis are presented in Chapter 9, "Dynamic Tessellation."

**Image quality.** Many other types of properties should be considered when determining the desired tessellation density. For example, if a control patch is positioned on the silhouette edge of a mesh, it will improve the image quality to tessellate the patch more than if it were perpendicular to the view direction. Another very good example of improving image quality with the use of varying tessellation levels is in the generation of dual-paraboloid shadow maps, or environment maps.[1]- The paraboloid projection is known to produce artifacts when it is generated with triangles that cover large areas of the paraboloid map. This happens because each vertex is transformed with the paraboloid projection, but the triangle primitives that the vertices are used to produce are linearly rasterized. This situation is depicted in Figure 3.22.

The situation becomes more problematic as a primitive covers more and more texels of the paraboloid map. A simple method to alleviate this issue is to calculate the screen-space area of a triangle in the patch constant function and then increase the tessellation factors according to this screen-space size. The tessellated points can then be properly projected in the domain shader stage to minimize the effects of this issue.

**Generic control patch calculations.** In addition to tessellation factors, the patch constant function can also produce user-defined attributes that will be applied to all tessellated points from the current control patch. This means that any data that can be shared by all

[5] Further details about paraboloid maps can be found in Chapter 13, "Multithreaded Samples."

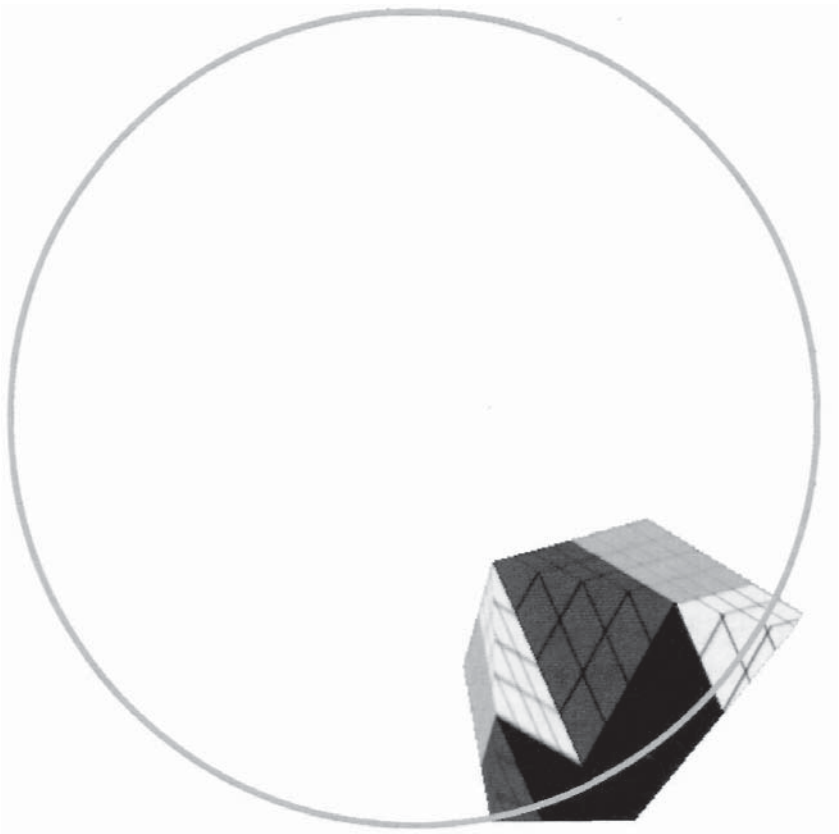


Figure 3.22. Artifacts from linear rasterization when used in projections that don't preserve lines.

control points of a control patch (such as a complete patch normal vector or a screen-space derivative of the patch position) can be calculated once and used many times later. Once again, this reduces the number of computations required to perform a calculation.

### 3.5.4 Hull Shader Pipeline Output

We have already seen that the results of executing the hull shader program are the output control points of the control patch to be used in tessellation. These control points are its primary way to pass information down the pipeline. Ultimately, these control points will be consumed by the domain shader stage, where they will be used to generate positions for each of the newly tessellated points from the tessellator stage. Requirements for the attributes of the output of the hull shader are very flexible. There are no mandatory system values that need to be written. Instead, it is up to the developer to supply the information needed for processing in later stages. But although the output requirements are very flexible, some common attributes are usually included. The positions of the control points are

typically provided by the hull shader program, which are typically either copied from the input control points or extracted from them. In addition, any material properties not read out of a resource would normally be included in the control points. These may include per control point colors, shininess, or material identifiers, just to name a few.

In contrast, the patch constant function's output is somewhat more predefined. It must produce the appropriately sized set of tessellation factors for use by the tessellator stage. These are written to output attributes with the system value semantic of SV_TessFactor and SV_InsideTessFactor. In addition, any patch constant information is also passed as output attributes. These attributes are not sent to the tessellator stage, since it doesn't perform custom processing, but are instead passed directly to each invocation of the domain shader program as input attributes.

## 3.6 Tessellator

The tessellator stage is the fixed function pipeline stage of the tessellation system. It is located somewhere between the hull shader stage and the domain shader stage, and this location defines the source and destination of its input and output. When the tessellation stages are used in the rendering pipeline, they are all used together—they cannot operate individually. As such, the tessellator stage plays an important role in implementing any tessellation algorithm. The location of the tessellator stage is highlighted in Figure 3.23.

The job of the tessellator stage is to convert the requested amount of tessellation, which is determined in the tessellation factors of the hull shader stage patch constant function, into a group of coordinate points within the current "domain." The domain in which these points are generated can be an isoline, triangle, or a quad—whichever is specified in the domain function attribute of the hull shader program. This should be somewhat surprising, since the operation described above makes no use of control points or control patches. Instead, the tessellator stage simply generates a set of points that indicate where vertices should be created within the specified domain. These points are later passed to the domain shader stage, where they are rejoined with the control points produced by the hull shader

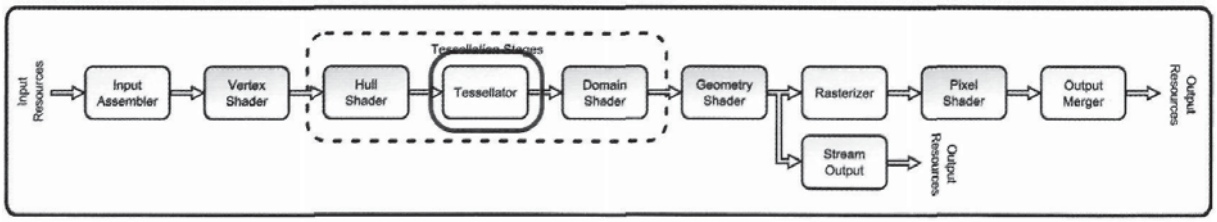



Figure 3.23. The tessellator stage.

program to give a physical meaning to the points. Understanding how the tessellator stage operates, and why it is implemented in this counterintuitive way, will provide a solid foundation upon which to build tessellation algorithms.

### 3.6.1 Tessellator Stage Pipeline Inputs

To understand the tessellator stage, we will begin by considering the type of data that flows into it. The vertex shader stage receives assembled vertices/control points from the input assembler stage, and the hull shader stage receives control patches created from the output control points from the vertex shader stage (along with primitive information from the input assembler). This basic concept is depicted in Figure 3.24.

In both of these cases earlier in the pipeline, the data that flows into the stage is processed in some fashion and then passed to the next stage. This is not the case in the tessellator stage. Instead, it receives the tessellation factors produced by the hull shader stage patch constant function. These factors are simple floating point numbers that indicate how heavily to tessellate the different parts of the domain object. Regardless of how the tessellation factors are calculated in the hull shader stage, if two control patches produce the same set of tessellation factors, they appear identical to the tessellator stage. Since the tessellator stage cannot distinguish between the two sets of input, it will produce precisely the same output in both situations.

Strictly speaking, all of the other pipeline stages would produce the same output if given two sets of identical input. The reason the tessellator is unique is that it is more likely to encounter the same input values from multiple different control patches, due to the "flattening" of the input data to simple factors. This is a profound difference from the other pipeline stages. This will be important to consider as we further explore the type of processing performed in the tessellator stage.

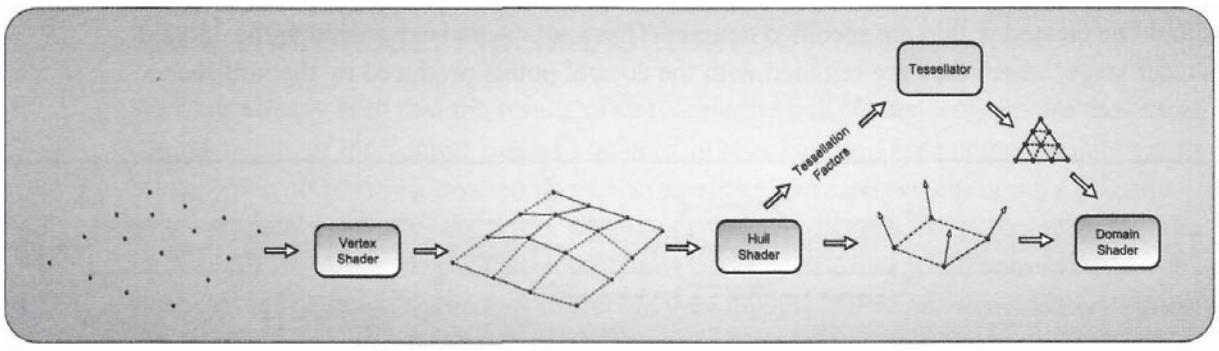



Figure 3.24. A representation of the data flowing through the pipeline.

## 3.6.2 Tessellator Stage State Configuration

The tessellator stage is not directly configured by the application, but by the attributes specified in the hull shader program (which *is* configured by the application). The available set of configurations is the collection of function attributes mentioned in the "Hull Shader" section. Since most of these function attributes are used to configure the tessellator stage (even though they are specified in the hull shader stage), we will take a closer look at what they are intended to do.

### Domain Attribute

We begin with the domain attribute. We have already mentioned that the tessellator stage does not actually tessellate control patches, but tessellates a *domain* object. The type of domain specified in this attribute can have one of three values: an `isoline`, `tri`, or quad. At a high level, we can say that the tessellator stage determines points within this domain that need to be realized by the domain shader stage. This attribute is mandatory, since the type of domain is also used to validate the number of tessellation factors produced by the patch constant function.

### Partitioning Attribute

The next attribute is the partitioning attribute. This can take on one of four values: `integer`, `fractional_even`, `fractional_odd`, or pow2. Each value identifies a different scheme for how to split up the specified domain. As we will see in more detail in Chapter4, "The Tessellation Pipeline," there is a significant difference in the chosen tessellation points for each of these partitioning schemes. This attribute is also mandatory.

### Output Topology Attribute

Once the specified domain is split up according to the specified partitioning scheme, the tessellator stage also needs to know how to assemble primitives from these individual points. The available options for this attribute are `triangle_cw`, `triangle_ccw`, and `line`. This attribute is also mandatory; otherwise, the primitives produced by the tessellation stage could have significantly different results than intended. For example, if the incorrect triangle winding were produced, the rendered geometry would appear inside out!

### Max Tessellation Factor Attribute

The final attribute that configures the tessellation stage is `maxtessfactor`. This attribute is an optional hint to the driver about the maximum possible amount of data amplification. Using this upper limit to the tessellation factors, the driver can efficiently preallocate enough memory to receive the results of the tessellation operations.

## 3.6.3 Tessellator Stage Processing

So what exactly is the operation that the tessellator stage performs with the tessellation factors and configurations mentioned above? The simple answer is that it selects a series of points within the given domain and then generates primitive data that creates an output primitive from these points later in the pipeline. However, the details about how this is performed and about how to control the produced output are not quite as clear. In this section we will take a closer look at the tessellation process and what is being performed by the tessellator stage.

### Sample Locations

The tessellator stage is more of a data generation stage than a data processing stage. Once a domain type has been specified, the inner tessellation factor and edge tessellation factors specify how much tessellation to perform. The tessellation itself occurs in two steps. First, tessellation points are chosen that will produce an appropriate number of triangles. If the edge tessellation factors are higher than the interior factors, more points will be chosen closer to the edges of the domain than in the middle. The inverse is also true—if a higher tessellation factor is received for the interior, more points will be chosen in the inner portion of the domain. This is depicted in Figure 3.25 for the quad domain.

The selected points are identified only by a set of coordinates within the current domain. At first, this is somewhat surprising when one considers what the tessellator stage is doing. These coordinates are the only data that the tessellator produces, and they only provide locations within a generic domain shape that should become vertices later, based on the control patch produced by the hull shader. This is what was meant by calling the tessellator stage a *data generation stage* instead of a *data processing stage.* It takes the domain configuration and the set of tessellation factors, and then generates all of the required sample locations. The amount of output data will typically be larger than the amount

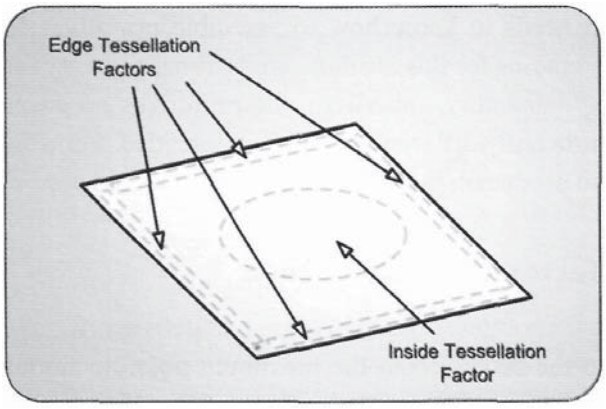



Figure 3.25. The regions of a quad that are affected by tessellation factors.

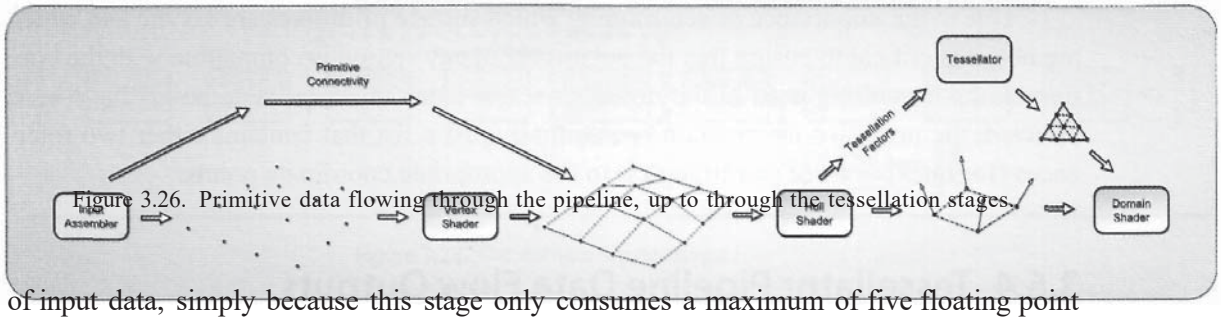



Figure 3.26. Primitive data flowing through the pipeline, up to through the tessellation stages.

of input data, simply because this stage only consumes a maximum of five floating point values for tessellation factors.

## Primitive Generation

The second step in the tessellator stage is to generate the required primitive information needed to use the sampled locations as renderable geometry later in the pipeline. You may wonder why the primitive data must be generated in this stage, since a primitive topology was specified in the input assembler stage. In fact, it is correct that the input assembler creates primitive connectivity information that skips the vertex shader stage and is passed to the hull shader stage when the tessellation system is active. However, when the tessellation stages are active, the input primitive from the input assembler stage must be one of the control patch types. This specifies the connectivity of the vertices as control points and defines a control patch, instead of a primitive type that can be rasterized directly. Figure 3.26 visualizes the distinction in how primitive data flows through the pipeline up to the end of the tessellation system.

We have already seen that the type of output primitive is specified by the outputtopology attribute. When lines are being generated, they have no front or back side, so it isn't important in what order the vertices are arranged in within the primitive. However, when triangles are produced, it certainly matters in which order the vertices are specified. As we will see in the rasterizer stage, triangles facing away from the current viewpoint are discarded and do not influence the final rendered image. Which direction a triangle faces is determined by taking a cross product of the two vectors created by the edges touching the first vertex. This is demonstrated in Figure 3.27.

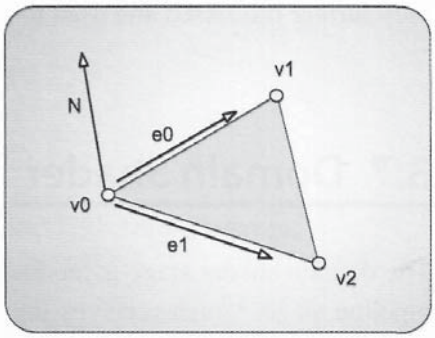



Figure 3.27. Determination of the direction a triangle is facing.

Due to the importance of determining which surface primitives are visible and which are not, it is critical to ensure that the outputtopology setting is compatible with the configurations for culling used in the rasterizer stage. Once the coordinate points have been selected, the primitive information is essentially just a list that contains either two references (for lines) or three (for triangles) to the appropriate coordinate points.

### 3.6.4 Tessellator Pipeline Data Flow Outputs

Once the tessellation process has been completed for a particular invocation of the tessellator stage, the results are passed through a system value semantic to the domain shader stage. The SV_Domain Location system value provides one domain coordinate point to the domain shader stage at a time, and the domain shader program is invoked once for each of these output points. Before the points are processed in the domain shader, they are simple coordinates in the current domain.

Since the primitive topology information created for the freshly tessellated geometry is not used by the domain shader stage, it is conceptually passed beyond the domain shader stage to the geometry shader stage. The geometry shader stage is aware of complete primitives, and hence can make use of the information. One important note to keep in mind is that there is *no* primitive topology with adjacency information available for creation in the tessellator stage. This means that when the tessellation stages are active, the geometry shader stage can't receive a primitive type with adjacency. This limits the available algorithms that can be implemented in the geometry shader program, but typically, any calculations that would require adjacency information can be incorporated into either the hull or domain shader programs, since they *do* have various levels of adjacency information available to them. If the geometry shader stage is not active, the primitive information and vertices created in the domain shader stage are passed directly to the rasterizer stage, where they further processed and used to generate pixels.

## 3.7 Domain Shader

The domain shader stage is the final stop in the tessellation system. It is a programmable pipeline stage, which receives input from both the hull shader stage and the tessellator stage, and produces output vertices, which can be further processed by the rasterization stages later in the pipeline. With this responsibility, the domain shader stage can be considered the heart of the tessellation algorithm, which is set up by the hull shader and by the granularity of the produced geometry set up by the tessellator stage. The location of the domain shader stage is highlighted in Figure 3.28.

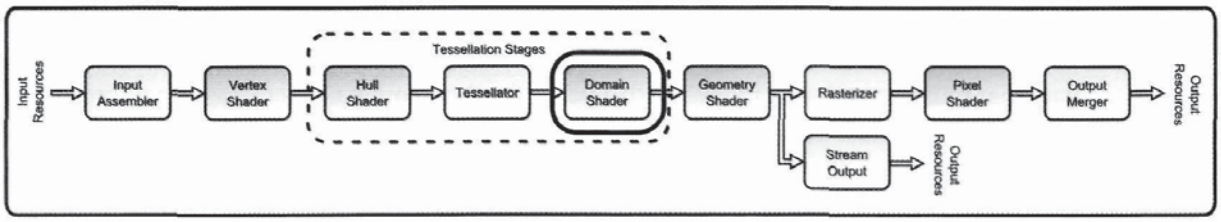



Figure 3.28. The domain shader stage.

## 3.7.1 Domain Shader Pipeline Input

The domain shader stage is required to produce vertices to complete the tessellation process and produce output geometry. It is invoked once for each coordinate point produced by the tessellator stage. To create these vertices, it receives the complete control patch produced by the hull shader stage. This can vary from 1 to 32 control points, but it must match the number of output points declared in the hull shader program's outputcontrolpoints attribute. The control patch is declared here in a similar manner to how it was used in the hull shader program, where the individual control points are provided in a template style attribute. The structure type for each control point is the first argument, and the number of points in the control patch is the second argument. Listing 3.15 provides a sample domain shader program input signature, which demonstrates this template style attribute.

```
struct HS_CONTROL_POINT_OUTPUT
{
    float3 WorldPosition          : POSITION;
};

Struct HS_CONSTANT_DATA_OUTPUT
{
    float Edges[3] : SV_TessFactor;
    float Inside   : SV_InsideTessFactor;
};

[domain("tri")]
DS_OUTPUT DSMAIN(
          const OutputPatch<HS_CONTROL_POINT_OUTPUT, 3> TrianglePatch,
          float3 BarycentricCoordinates : SV_DomainLocation,
          HS_CONSTANT_DATA_OUTPUT input )
{
    // ...
}
```

Listing 3.15. A sample domain shader declaration, demonstrating the input attribute declarations.

As seen in Listing 3.15, the input control patch is declared as a const input parameter, since it can't be modified. Due to this read-only access, any modifications or additional data that must be included in the control patch for the domain shader program to create the output vertices must be provided by the hull shader program before this stage.

In addition to the control patch, the domain shader program also receives the domain location that it is supposed to implement from the tessellator stage. This data is provided as the SV_Domainl_ocation system value semantic. The format of this attribute varies, depending on which domain type is specified, with a float2 used for `isoline` and quad domains, and a float3 for `tri` domains. These values represent where within the domain this domain shader invocation should generate vertex data. In this respect, the coordinate location defines a sampling point over a virtual surface determined by the function that the domain shader program implements. We will investigate this concept further in the processing portion of this section.

The final set of input data that the domain shader program receives is the constants that were produced by the patch constant function of the hull shader stage. The data contained within this structure varies, according to what is calculated in the patch constant function, but it will remain the same for each domain shader program invocation for a given control patch.

To provide a high-level view of what the domain shader program is required to do, we can examine its input from the pipeline. All of the input control patch data is provided to every invocation of the domain shader program. In addition, the data received from the patch constant function is also constant across a complete control patch. This means that the only input that varies between invocations of a domain shader for a complete control patch is the domain location. This makes sense, since the number of points produced by the tessellator stage will vary, but the control patch data input doesn't change with a varying tessellation level. The separation of these two functionalities in the hull shader stage provides a clear distinction in this regard.

## 3.7.2 Domain Shader State Configuration

As a programmable pipeline stage, the domain shader stage can use the standard set of configurations available to all programmable stages. Again, these configurations are set by the application through the ID3D11DeviceContext interface with the familiar method names, with the exception that the beginning of the method names use DS to indicate that they will affect the domain shader stage. Since these methods and their usage characteristics have already been discussed, we won't repeat the code listings here. Further details on using these methods can be found in the "Vertex Shader" section. The methods are listed here for reference.

ID3DllDeviceContext::DSSetShader()

ID3DllDeviceContext::DSSetConstantBuffens()

```
ID3DllDeviceContext::DSSetShaderResources()

ID3DllDeviceContext::DSSetSamplers()
```

## 3.7.3 Domain Shader Stage Processing

The primary job of the domain shader stage is to create vertices using the generated set of coordinate points from the tessellator stage, by using the control patch and patch constants produced by the hull shader stage. The most important task for creating these vertices is determining their position, since this will directly affect how the final tessellated geometry will be rasterized. With this in mind, we will take a closer look at how the position is calculated in the domain shader program and will provide some high-level concepts for thinking about this process.

The data held in the points within the control patch can represent a wide variety of different types of higher-order surfaces. The control points can be used to implement Bezier curved surfaces (Akenine-Moeller, 2002) just as easily as they can represent normal vertices, as used in curved point triangles (Vlachos, 2001).[16] The only requirements are that the hull shader program must produce the expected format of data, and that the domain shader must implement a compatible algorithm for generating an output position for each coordinate point.

In a more generalized view, we can think of the control patch as a set of parameters which define a virtual surface. This surface must be valid over the entire domain of the control patch, since this is the region in which all of the input coordinate points will reside. Then we can think of each of the coordinate points produced by the tessellator stage as a selected location where we want to sample the virtual surface. This can be visualized with a simplified case, as shown in Figure 3.29, where we see an isoline domain, its control points, and the virtual surface that it represents. The selected sampling points along the isoline domain indicate the locations that are sampled along the virtual surface.

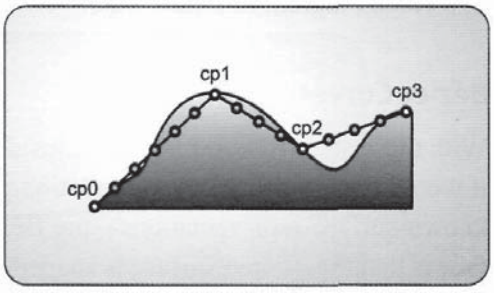



Figure 3.29. A depiction of how a control patch defines the parameters of a virtual surface, and the tessellator-generated points that can be considered as sampling locations for this surface.

The important message to take from this concept is that the virtual surface remains the same regardless of where the tessellator-generated points are located or how many there are. This can help us break the overall

[16] Bezier curved surfaces are discussed in more detail in Chapter 4, while curved point triangles are discussed further in Chapter 9.

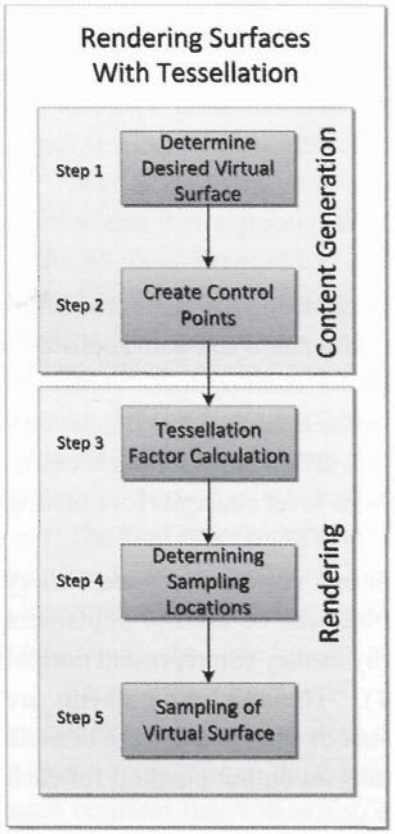



Figure 3.30. The process of constructing geometry from a virtual surface at the specified sampling points.

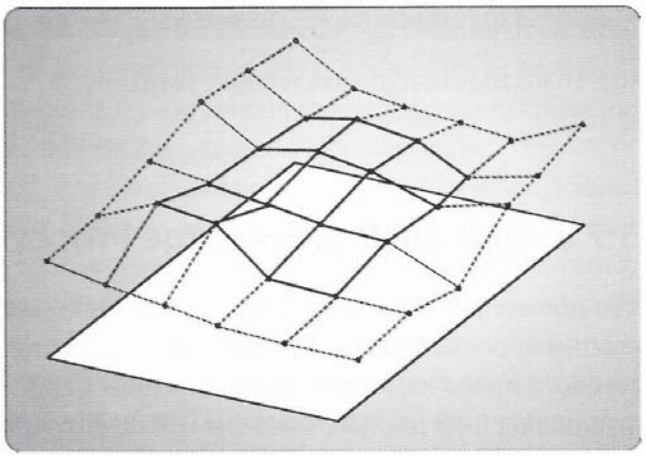


Figure 3.31. The set of control points that will affect a single quad of the surface.

tessellation algorithm design task into a few different components. First, our control patches must define the surface that we actually want. This is typically done at the content creation phase if the object is pre-created by an artist. Second, our tessellation factors must be calculated from the control patch to ensure that our desired surface is adequately sampled and therefore it appears correct in the current viewing conditions, while minimizing the number of vertices required to achieve that visual quality. Finally, when provided with a set of coordinates, our algorithm must be able to properly sample the virtual surface from the control patch data. These steps are shown in a block diagram in Figure 3.30.

## Bezier Curves

With these steps in mind, we can consider a traditional tessellation algorithm and see how it would fit into this whole tessellation paradigm. We will consider one of the most well-known surface representations—the Bezier surface. The general concept of a Bezier surface is that the desired surface is shaped by moving a grid of control points. The number of points can vary, but for our example we will assume a 4x4 grid of control points to define the surface shape for a quad portion of the surface. This setup is shown in Figure 3.31.

Once the surface has been edited into the desired shape, the control points are stored as vertices in a vertex buffer, and an index buffer is used to define groups of control points as control patches. Next, when the object is going to be rendered, the input assembler builds the control points as vertices and passes them to the vertex shader. At the same time,

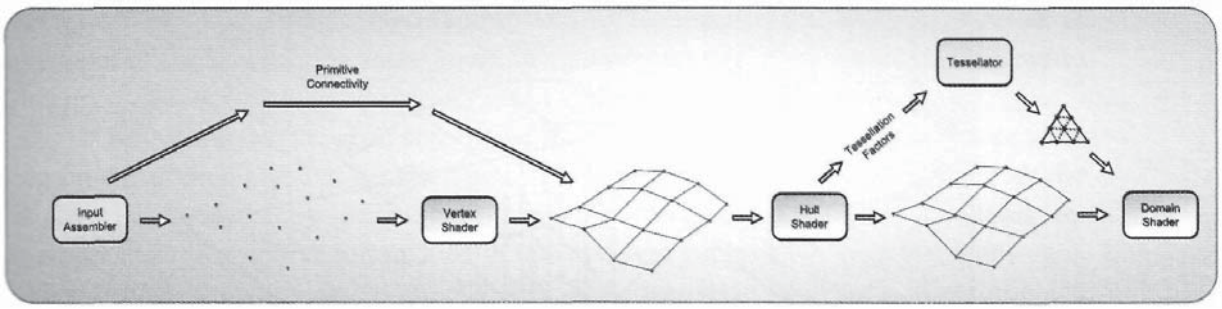



Figure 3.32. Our 4x4 Bezier surface as it is passed through the input assembler, vertex shader, and hull shader stages.

it determines which control points belong to a particular control patch, based on the indices provided in an index buffer and its primitive type setting (a 16-point control patch in this case). The control patch points are read by the vertex shader and passed to the hull shader. The 4x4 group of control points is then read by the hull shader and passed to the domain shader in the same 4x4 control patch configuration. The process up to this point represents step one in our block diagram from Figure 3.30. Figure 3.32 depicts this process.

Next, the patch constant function must calculate the needed tessellation amount to sufficiently represent our Bezier surface in the given viewing conditions. This can take into account the size of the control patch, the distance from the current view point, or any number of other metrics. The required tessellation factors are passed to the tessellation stage, which converts that data into a series of coordinates within the domain. Those points are then passed to the domain shader stage. This represents step two in our tessellation process, and is depicted in Figure 3.33.

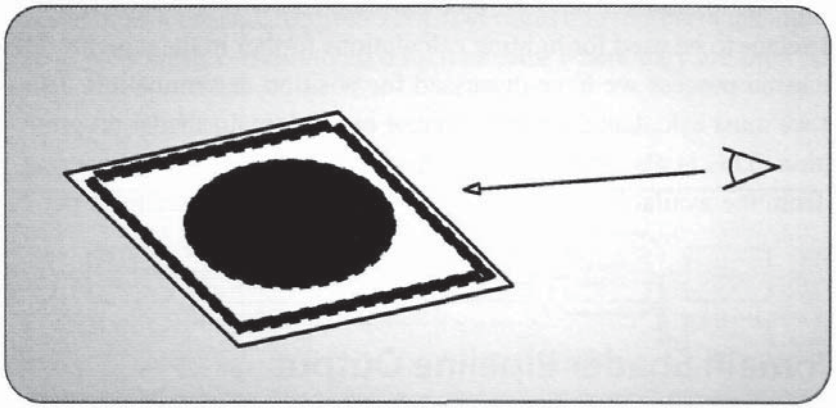


Figure 3.33. Determining the required edge factors for a particular quad, based on the current viewing conditions.

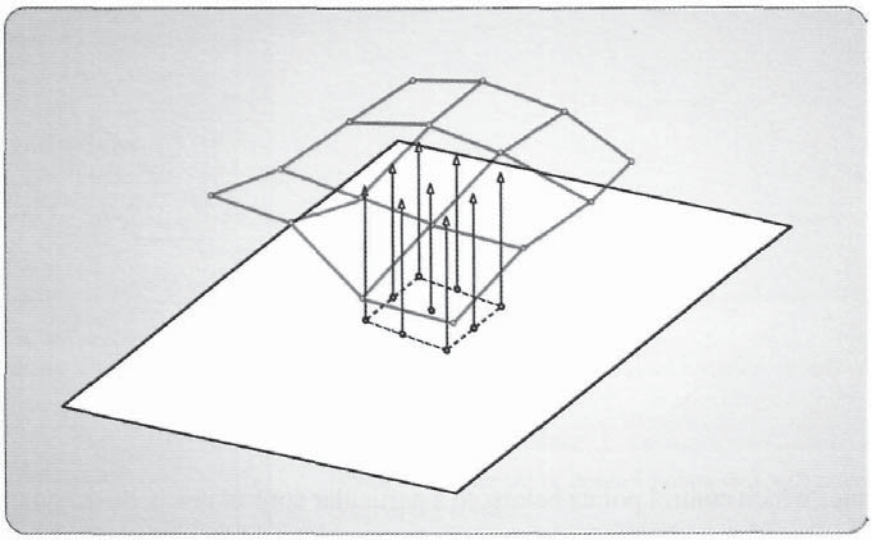


Figure 3.34. The generation of vertex locations in the domain shader for our 4x4 Bezier surface example.

Finally, the domain shader program must be able to receive the control patch, along with one coordinate point for each invocation of the domain shader, and produce a vertex that represents the Bezier surface. This is basically performed by evaluating the equations for Bezier surfaces, where each of the control points contributes to the calculated location of the vertex. The amount of the contribution is determined by the coordinates, which more or less specify the proximity to each of the control points where we are evaluating. Once this location has been calculated, the position is produced by the domain shader and is passed down the pipeline for further processing. This is shown in Figure 3.34.

Of course, the domain shader will normally need to calculate more than just the position of each of the tessellated vertices. One attribute that will be required most of the time is a normal vector to be used for lighting calculations further in the pipeline. However, this follows the same process we have discussed for position determination. The only difference is that we must calculate the normal vector in the domain shader program, in addition to the position. This is also true for any other attribute needed for rendering. It must be calculated from the available information, even if it is simply read from per-control point attributes.

### 3.7.4 Domain Shader Pipeline Output

After the required attribute data is calculated, the newly created vertices are returned by the domain shader program and passed to the next stage. The position output from this stage

is typically written in the SV_Position system value semantic, which must be present in the input to the rasterization stage. In addition, any other per-vertex attributes required to determine the final pixel color are also added to the domain shader output.

It is possible that the geometry shader stage can be either active or inactive, depending on the desired pipeline configuration. If it is active, output from the domain shader is passed to the geometry shader stage, where it is consumed as complete primitives (remember that the primitive information in this case was generated at the tessellator stage). Otherwise, the output is passed directly to the rasterizer stage, where it is also consumed as primitives.

## 3.8 Geometry Shader

The geometry shader is the final pipeline stage that can manipulate the geometry being passed through the pipeline before it is rasterized. It is a programmable stage and has several unique capabilities not found in any other stage, including the ability to programmatically insert/remove geometry in the pipeline, the ability to pass geometry information to vertex buffers through the stream output stage, and the ability to produce a different primitive type than is passed into it. This provides some very interesting use cases for this stage, which are outside of the traditional pipeline model, ranging from saving processed geometry data to a file, pipeline operation debugging, and of course, rendering operations. The geometry shader stage also operates on complete primitives, including adjacent primitive information, which provides the additional ability to analyze and test various aspects of each primitive and perform customized calculations, depending on the geometry and its immediate neighbors. The location of the geometry shader stage in the pipeline is highlighted in Figure 3.35.

The geometry shader stage receives a list of vertices that represent the input primitives. It is then free to pass these vertices to an output stream, where they are then re-interpreted

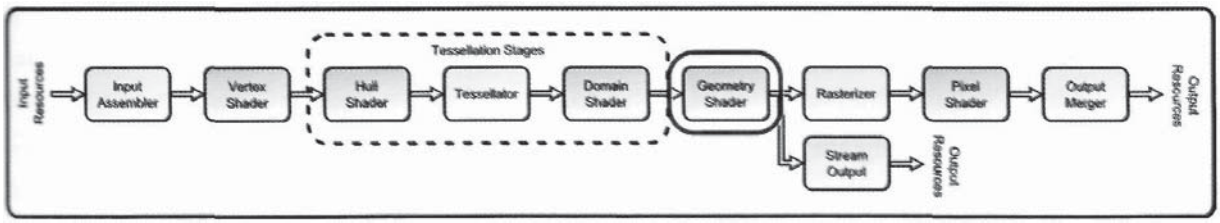



Figure 3.35. The geometry shader stage.

as primitives as they are passed to the next stage in the pipeline. Depending on the type of stream object declared for receiving this output, the streamed vertices will be used to construct different primitives. In addition, up to four output streams can be declared to use in conjunction with the stream output stage, making it quite simple to produce geometry intended for rasterizing as well as geometry to save into a buffer resource. This streaming model is the heart of the operation of the geometry shader, and we will see throughout this section how it can be used in various situations.

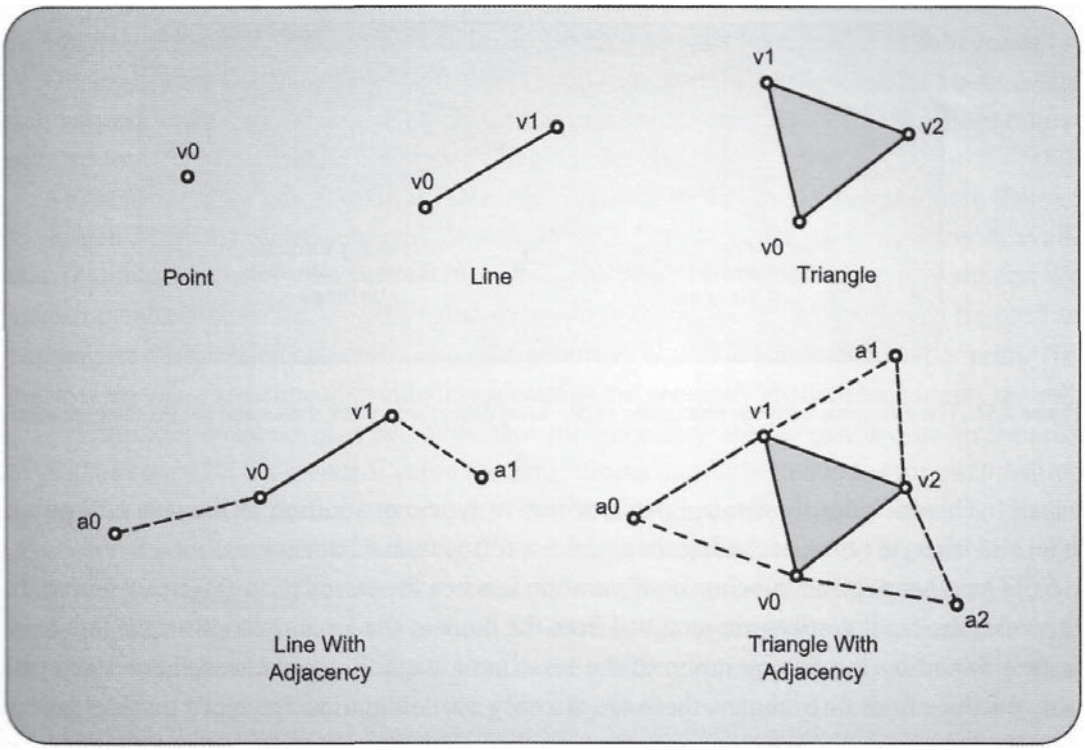
## 3.8.1 Geometry Shader Pipeline Input

The geometry shader stage operates on complete primitives composed of an array of vertices. Depending on which type of primitive is being passed, the array will vary in size, from a single vertex all the way up to six vertices for a triangle with adjacency information. The input to the geometry shader program declares the type of primitive that it will receive in the input attribute definition. A sample geometry shader function signature is shown in Listing 3.16.

```
[instance(4)]
[maxvertexcount(3)]
void GSScene( triangleadj GSSceneIn input[6],
              inout TriangleStream<PSSceneIn> OutputStream )
{
    PSSceneIn output = (PSSceneIn)0;

    for ( uint i = 0; i < 6; i += 2 )
    {
        output.Pos = input[i].Pos;
        output.Norm = input[i].Norm;
        output.Tex = input[i].Tex;

        OutputStream.Append( output );
    }

    OutputStream.RestartStrip();
}
```

Listing 3.16. A sample geometry shader program, demonstrating how to declare an input stream attribute.

In this listing, the input attribute represents a triangle with adjacency, which produces four triangles out of six vertices. The order of the vertices within this input array depends on which type of primitive is being passed to the function. The various primitive types, their vertex counts, and the order that their vertices are provided in, are listed in Table 3.1. In addition, each of these primitive types is shown in Figure 3.36 for easy reference.




-igure 3.36. The various primitive arrangements that are available to be received by the geometry shader.

| Keyword | Primitive Type | Number of Vertices | Vertex Ordering |
|---|---|---|---|
| point | point list | 1 | [V0] |
| line | line list, line strip | 2 | [V0,V1] |
| triangle | triangle list, triangle strip | 3 | [V0,V1,V2] |
| lineadj | line list w/adj., line strip w/adj. | 4 | [A0,V0,V1,A1] |
| triangleadj | triangle list w/adj., triangle strip w/adj. | 6 | [V0,A0,V1,A1,V2,A2] |

Table 3.1. A table for determining how to evaluate the geometry shader input vertex stream.

With the primitive information available to it, the geometry shader program can inspect the geometry, perform visibility tests, or calculate high-level information for the primitive as a whole. There are two different possible pipeline configurations that can send primitive data to the geometry shader stage. If tessellation is inactive (the hull and domain shader programs are set to NULL), the input vertices are received directly from the vertex shader, with the primitive connectivity specified by the input assembler. In this case, the primitive topology can be any of the primitive topologies that the input assembler can

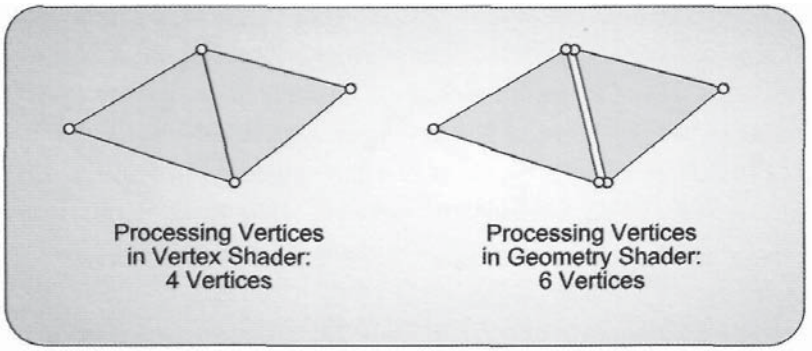



Figure 3.37. The difference between processing vertex-level data in the vertex shader and the geometry shader.

specify. This includes the control patch primitive types, in addition to the standard point, line, and triangle types, including their various forms with adjacency.

The other possible pipeline configuration is when the tessellation stages are active. In this case, the input vertices are received from the domain shader and the primitive topology is determined by the configuration of the tessellator stage. Since the tessellator stage can only produce lines or triangles, these are the only available primitive types that the geometry shader can receive when tessellation is active. This is why primitives with adjacency are not supported when the tessellation stages are active.

Regardless of the pipeline configuration, the geometry is received as a list of vertices representing the primitives present at this point in the pipeline. This has some performance implications, since any of the rendering modes that reduce the amount of vertex processing will not produce the same savings in the geometry shader. For example, if a triangle strip with four vertices is processed in the vertex shader, four invocations of the vertex shader will process the complete set of geometry. However, if each vertex is processed in the geometry shader instead, all three vertices will be submitted to each geometry shader invocation—which is the number of primitives being processed. In this case, there would be two geometry shader invocations, and each would receive the three vertices of the triangle being processed. Therefore, processing the vertices in the vertex shader requires only four sets of calculations, while the same operations performed in the geometry shader will require six. This difference is shown in Figure 3.37.

We can also see from Listing 3.16 how a stream output object is declared for a geometry shader. This object is actually the mechanism that the geometry shader uses to pass its output out of the stage, as opposed to the function returning its output values directly. Even so, the stream is passed as an input argument and hence will be discussed in this input discussion. The syntax for declaring a stream output object begins with the `inout` keyword, which specifies that the object is both an input and an output. This is followed by one of three possible stream types. The stream output object can be used to output a point list, a line strip, or a triangle strip; each of these is represented by the available

stream declarations. The argument can be declared as a PointStream<T>. LineStream<T>, or TriangleStream<T>, where each T represents the format of the vertex structure that will be passed into the stream. We will further discuss the mechanics of how these output streams must be used in the "Geometry Shader" section.

One final input consideration is the use of two system value semantics, one that we have seen before and one new one. The SV_PrimitiveID system value attribute is available to uniquely identify each of the primitives that are passed to the geometry shader. We have previously seen this system value in the hull shader stage, where it can be used to uniquely identify control patches. Since the geometry shader is invoked once per primitive, this system value semantic also uniquely identifies the geometry shader invocations, as well.

The other system value semantic that the geometry shader can use as an input is SV_GSInstanceID. This system value semantic works similarly to SV_InstanceID, which we saw in the vertex shader stage, except that the instances are actually instanced in the geometry shader stage itself, instead of in the input assembler stage. It is possible to declare a static number of instances to create for each primitive that the geometry shader receives. The SV_GSInstanceID is used to differentiate among the individual primitive instances and take the appropriate action in the shader program. This system value must be declared as a standalone input to the geometry shader, since it can't be declared in one of the input streams. Since the identifier indicates which geometry shader invocation is being processed, it makes sense to not allow this to be a per-vertex streaming attribute. The technique to enable this instancing mechanism is discussed in the "Geometry Shader State Configuration" section, while the use of the functionality is discussed in more detail in the "Geometry Shader Stage Processing" section.

## 3.8.2 Geometry Shader State Configuration

Like all of the other programmable pipeline stages, the geometry shader stage provides the usual group of common shader core resource configurations that can be manipulated by the application with the ID3DllDeviceContext interface. There are some key differences in how geometry shader objects are created, due to the geometry shader stage's close relationship to the stream output stage, which we will cover briefly in this section, with a more detailed examination in the "Stream Output" stage section.

### Geometry Shader Program

The geometry shader object can be created with one of two methods. Which one must be used depends on whether the stream output stage will be used or not.[17] If the stream output stage will not be used, the geometry shader object is created in the same way that we have

[17] The particular details of how the geometry shader writes data to the stream output stage is covered in the "Geometry Shader Stage Processing" section.

already seen for the other programmable stages using the ID3DllDevice: :CreateGeometry Shader(…) method. However, if the stream output stage will be used, the shader object must be created with the ID3DllDevice: :CreateGeometryShaderWithStreamOutput(...) method. This method takes several additional parameters, which configure the stream output stage and the way that data will be streamed to the buffers attached to it. The details of how to configure these parameters for the stream output version will be discussed in the "Stream Output" stage section of this chapter. Simply keep in mind for now that there are two different ways to create the shader object, depending on whether the stream output will be used.

Once the geometry shader object has been created, it can be set in the geometry shader stage with the usual ID3DllDeviceContext method. Similarly, the constant buffers, shader resource views, and samplers required for the geometry shader program are manipulated in the standard common shader core methods. They are listed here for reference.

ID3D11DeviceContext::GSSetShader()

ID3D11DeviceContext::GSSetConstantBuffers()

ID3D11DeviceContext::GSSetShaderResources()

ID3D11DeviceContext::GSSetSamplers()

### Function Attributes

The geometry shader stage also supports two function attributes that must be declared prior to the geometry shader function in the HLSL source file. The first attribute is the maxvertexcount parameter, which allows the developer to specify just that—the maximum number of vertices that an invocation of the geometry shader will emit into its output stream. This is required to ensure that the geometry shader doesn't output an erroneous number of vertices if it has a logic error. It also lets the GPU properly allocate memory for the number of vertices it expects to be produced by each invocation of the geometry shader. This attribute is mandatory and must be provided for the geometry shader function to compile properly. An example of this function attribute can be seen in Listing 3.16.

There is a limit to how much data that can be produced by a single geometry shader invocation. The number of scalar values may not exceed 1024. This means that the number of scalar values used in a vertex structure must be summed, and then multiplied by the maximum number of vertices provided in the maxvertexcount attribute. This value must be less than or equal to 1024. We will see later that up to four different streams can be used simultaneously when using the stream output functionality, but for the purposes of this maximum output calculation, we can simply take the largest vertex size of all of the output streams being used and multiply that by the maxvertexcount attribute.

The second function attribute allowed by the geometry shader is the instance attribute. This attribute activates the geometry shader instancing mechanism, in which the primitives passed into the geometry shader are duplicated as many times as specified in the instance attribute declaration. The geometry shader can then declare the SV_GSInstanceID (as described above) to provide a unique identifier for the instance. The maximum number of instances that can be created in this way is 32, which allows for significant data amplification. This instancing technique is aimed at simplifying geometry shader programs that need to process geometry for several different outputs, such as when more than one viewpoint is being rendered simultaneously. For example, when generating a cube map[18] with a single rendering pass, geometry shader instancing can be used to generate the six copies of the primitive data, which can then choose the appropriate transformation matrix with the SV_GSInstanceID system value semantic.

## 3.8.3 Geometry Shader Stage Processing

### Geometry Shader Process Flow

Before we investigate the possible operations that can be performed in the geometry shader, we must first clarify how it uses the stream objects to produce its output. An example geometry shader was shown in Listing 3.16, and we will reference it throughout this discussion. As we have seen in the previous sections, the geometry shader program receives input primitives as an array of vertices. The ordering of the vertices will vary, depending on the type of input primitive that the pipeline produces prior to the geometry shader stage, as shown in Table 3.1 and Figure 3.36, respectively.

Once the vertex data is available within the geometry shader, some calculations are performed to either modify the input vertices or create entirely new ones. The sample code in Listing 3.16 receives a triangle with adjacency as its input, and then simply passes the main triangle through to its stream output object, ignoring the adjacency information. The method used to generate the output primitives is to call the stream output object's Append () method, which takes an instance of the output vertex structure as its argument. With three different stream types available (one for triangle strips, one for line strips, and one for point lists) a series of vertices that are appended to a stream will create different primitives. For example, if a triangle stream is used, and 5 vertices are appended to it, then a total of 3 triangles will be created in the "strip" vertex.[19] Each additional vertex will generate a new triangle using the previous two vertices to complete the primitive. The same methodology exists for the line strip stream object, which would create an additional line primitive for each additional vertex appended after the first one.

[18] Cube maps are briefly described in the "Using 2D Textures" section of Chapter 2, "Direct3D 11 Resources."

[19] This is the same "strip" ordering principle that was explained in the "Input Assembler" section of this chapter.

If the triangle/line strip needs to be restarted to create detached groups of geometry, the geometry shader can call the RestartStrip() method of the output stream. This will reset the primitive generation to allow another strip to be started, which is not connected to the previously streamed geometry. While it is more efficient to produce geometry in a single strip, if the required output geometry must reside in multiple disconnected sections, then the strip must be restarted. This process of producing primitives by appending vertices to the output stream continues until the geometry shader has completed, which marks the end of this particular invocation of the shader program.

### Multiple Stream Objects

One of the new features in Direct3D 11 is the ability of the geometry shader program to use up to four different stream output objects. When properly configured, these objects can be used not only to pass pass primitive data to the rasterizer stage, but to send their data to the stream output stage, where it can be stored in buffer resources. Since the buffer resource is accessible to the application,[20] this means that the stream of pipeline data is directly available for reading or storing in a file. This is a powerful facility, which can be used for some interesting data analysis scenarios. To declare multiple streams in the geometry shader program, you simply include additional stream objects as input parameters to the function. Listing 3.17 demonstrates this technique. These streams are used in the same way that the individual output stream is, with the AppendQ and RestartStripQ methods. Each output stream can be written to independently from one another—they don't have to have the same output frequency.

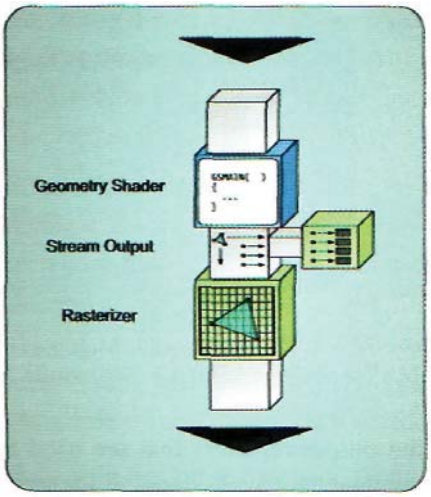
```
void MyGS( InVertex verts[2],
           inout PointStream<OutVertex1> myStream1,
           inout PointStream<0utVertex2> myStream2 )
{
    OutVertex1 myVert1 = TransformVertex1( verts[0] );
    0utVertex2 myVert2 = TransformVertex2( verts[1] );
    myStream1.Append( myVert1 );
    myStream2.Append( myVert2 );
}
```

Listing 3.17. A geometry shader that declares more than one output stream, with different vertex structure types.

Normally, when the geometry shader uses only a single output stream, the results of that stream are passed on to the rasterizer stage, where they are processed and split into

[20] Techniques for manipulating resources, including how to read their contents from C/C++, is provided in Chapter 2: Direct3D 11 Resources.

fragments. When there are multiple streams used in the geometry shader program, only one of the declared streams can be selected to be passed on to the rasterizer stage through the stream configuration set by the application when compiling the geometry shader program. The remaining streams must be configured to be streamed to a buffer (one buffer per stream) attached to the stream output stage. The stream that is passed to the rasterizer can optionally be streamed to a buffer as well. Details about how to configure these streams from the application will be covered in the "Stream Output" stage section. Figure 3.38 shows multiple stream outputs from the geometry shader.




Rgure 3.38. Multiple stream outputs from the geometry shader.

A very important consideration when using multiple output streams is that all stream types must be point streams. This means that all vertices passed to a buffer resource are collected individually, which also means that it is not possible to produce triangles or lines that share vertices. In this scenario, a given set of geometry may consume more memory than would normally be used with line strips or triangle strips. However, if the algorithm in the geometry shader produces a known pattern of vertices, it may be possible to use a preloaded index buffer to reference the vertices as the desired primitives. In addition to this restriction, if a stream is sent to the rasterizer stage, it will only appear as points. Because of this, most geometry shader programs with multiple output streams don't rasterize any of the streams unless point rendering is desired.

Another possible configuration is to use a single stream output object (which can use any of the three stream types) and pass it to both the rasterizer and the stream output stage. Data sent to the stream output buffer is still stored as non-indexed geometry, so it causes the same memory consumption increases as mentioned above. However, if the object is being rasterized, it is possible to use triangles and lines instead of just points. These are all options for the developer, and each situation may require using a different combination of these streams to be used.

The final consideration when using multiple output streams is the scalar value output limits imposed on each geometry shader invocation. We have already seen that there is a limit of 1024 scalar values, and that this limit applies to the total number of scalars passed into an output stream. In the case of multiple streams, we take the maximum vertex size and use this to determine the maximum number of vertices that can be passed into the output streams. There is only one set of output registers in the stage, and the registers are shared among all of the output streams. The maximum vertex size is chosen, since

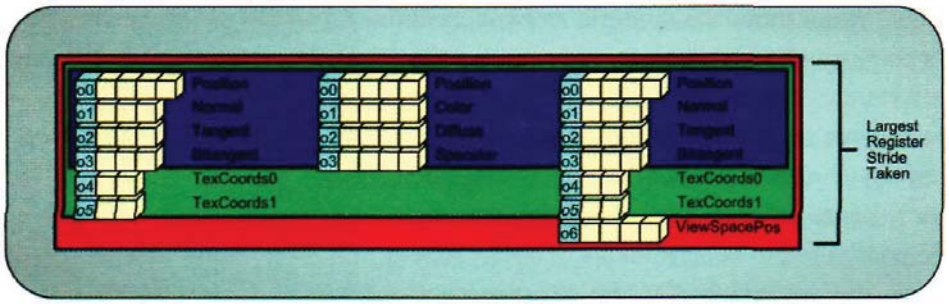



Figure 3.39. Multiple vertex structures using the same output registers.

the output registers that are used to receive the output vertex data will overlap with one another. This is depicted in Figure 3.39.

Since the output frequencies of the streams are not required to be equal, there is no easy way to ensure that dynamic branching won't produce an execution path that streams out the largest vertex for all vertex outputs. Therefore, the easiest way to ensure that only 1024 scalars are output is to determine at compile time the "worst case" vertex output and only allow a valid number of those vertices to be streamed. Any other vertices that are streamed out will only reduce the total number of scalars produced, ensuring that the limit of 1024 scalars is observed.

### Primitive Manipulations

Another interesting result of the geometry shader's process flow is that while primitives must be passed into the stage, they may or may not be passed to the next stage. This lets the geometry shader selectively discard unneeded geometry, reducing the amount of geometry for the rasterizer to process, and allowing for special rendering operations that require only a subset of a model to be rendered. A good example of this is the rendering of a model's silhouette to indicate that it is highlighted.

### Shadow Volumes

Shadow volumes are a shadowing technique that extracts a model's silhouette as viewed from a light source. The extracted silhouette is expanded outward away from the light, and this expanded geometry is rasterized into the stencil buffer[21] as an indication of which pixels are inside of a shadow volume. This is the heart of the algorithm—the extracted silhouette geometry forms a volume. If the viewer only sees one side of the shadow volume at a particular pixel, it means that that particular point in the scene is inside of the shadow and would not be illuminated by the light. If the viewer can see both sides

[21] The stencil buffer and its operation are described later in this chapter, in the output merger stage section.

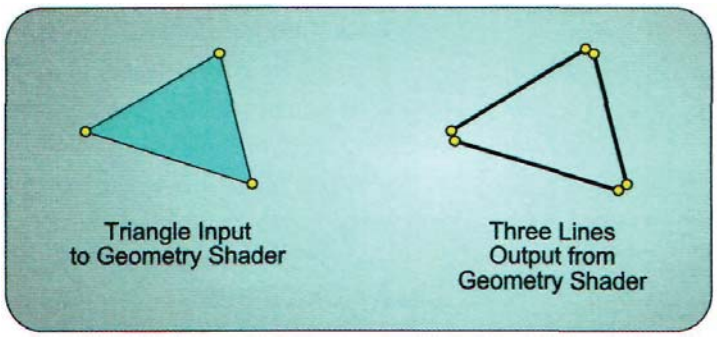



Figure 3.40. Splitting a triangle into three lines to provide a customized wireframe rendering.

of the shadow volume at that pixel, then that point in the scene lies on the far side of the shadow volume and hence should be illuminated.

Before the geometry shader existed, shadow volume extrusion had to be performed on the CPU and then passed to the GPU to be rasterized into the stencil buffer. However, the geometry shader can easily detect when a triangle is located at the silhouette by comparing its face normal vector to the adjacent triangle face normal vectors. If the current triangle is facing the camera, and at least one adjacent triangle is facing away from the camera, then the edge that the two triangles share should be extruded as a part of the shadow volume. If a triangle is not a portion of the silhouette, the geometry shader can discard it by simply not outputting its vertices. In this way, the geometry shader reduces the workload downstream in the pipeline by eliminating unneeded geometry from further processing.

## Point Sprites

Since the geometry shader receives primitives in its input array of vertices and it can declare the type of output stream it wants, there are no restrictions regarding the input and output primitive types. In combination with the geometry shader stage's ability to produce a variable number of vertices, this allows for complete control over primitive type conversion. For example, a triangle passed into the geometry shader can be converted to three lines representing the edges of the triangle. This is a simple way to implement a wireframe rendering scheme. The creation of the individual triangle edges is shown in Figure 3.40.

Another popular primitive conversion technique is to convert point primitives into a pair of triangles that form a quad. These quads can then have a texture applied to them, which effectively converts point geometry into a sprite. This process is typically referred to as *creating point sprites,* and is frequently used to give particles in a particle system a more compelling appearance. The process of creating point sprites is demonstrated in Figure 3.41. An example of this technique is used in Chapter 12 to add textures to the particles in the particle system example.

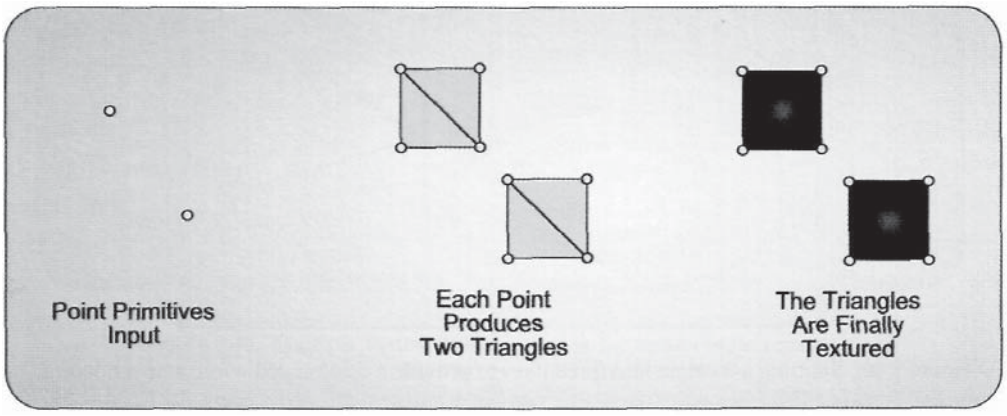



Figure 3.41. The generation of point sprites from point primitives.

### Instancing Geometry

The final feature of the geometry shader that we will examine is its ability to perform instancing. As mentioned above, the number of instances of each primitive to create is statically declared with the `instance` attribute. The individual instances can then be identified by the SV_GSInstanceID system value semantic. This is a fairly generic ability, which can be applied to many different situations. For example, one point sprite can be instanced and offset from another to amplify the number of particles that appear in the final rendering, while keeping the number of particles in the system to a smaller amount.

Another good example of how to use this functionality is to simultaneously create geometry for multiple render targets. This is especially effective when the geometry for two render targets must have different transformation matrices applied to them, as is the case in cube-mapped environment mapping or dual-paraboloid environment mapping. Separate instances of a primitive are created, which can be transformed and sent to the appropriate render target. This would eliminate repeating all the processing that would have been required prior to the geometry shader stage if two full rendering passes were needed.[22]

## 3.8.4 Geometry Shader Pipeline Output

We have discussed many ways to use the output vertex streams of the geometry shader, and noted that these output streams are the mechanism by which the geometry shader passes its results down the pipeline. However, we have not discussed in detail what the vertices that are appended to the output streams can contain. Since the geometry shader stage occurs immediately before the rasterizer stage, it must produce vertices with the SV_Position

[22] This geometry shader instancing technique is implemented in the sample program for Chapter 13, which uses the dual-paraboloid environment map method.

system value semantic. In addition, the contents of this attribute must contain the post-projection clip space position of the vertex in order to be rasterized properly. Stages prior to the geometry shader stage could use this attribute with other forms of the position (such as object space, world space, or view space positions), but the geometry shader must pass the final clip space position to the rasterizer stage.

In addition to the position of each vertex, the geometry shader can take advantage of a pair of system values that have not been available prior to this stage in the pipeline. The first of these system values is SV_RenderTargetIndex. This system value provides an unsigned integer value that identifies the slice of a render target that this primitive should be rendered into. As described in Chapter 2, it is possible to create 2D texture resources that contain an array of textures. Each of these textures is referred to as a slice of the texture. If a render target is bound for output to the pipeline and contains multiple slices, then the geometry shader can dynamically decide which render target to apply a particular primitive to. This method of selecting the render target to be applied to goes hand in hand with the single-pass environment mapping examples described earlier in this section. The SV_GSInstanceID value would be used to determine which copy of a primitive is passed into each render target by copying its value into SV_RenderTargetIndex. Of course, the bound render target must have an appropriate number of slices to accommodate the range of instances specified in the geometry shader. We will discuss binding render targets to the pipeline in more detail in the "Output Merger" stage section. The use of this system value semantic is depicted in Figure 3.42.

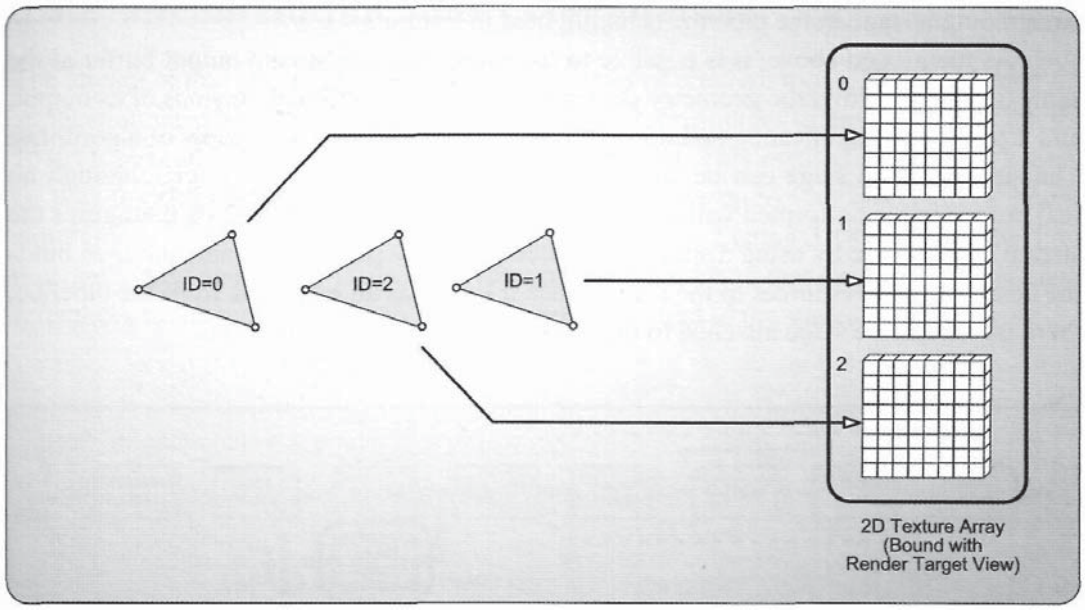



Figure 3.42. Using the geometry shader to write to multiple texture slices simultaneously with the "SV_RenderTargetIndex" system value semantic.

The second system value that the geometry shader can use is the SV_Viewport ArrayIndex attribute. This attribute operates in a similar way to SV_RenderTargetIndex, except that it determines which viewport it will be applied to, instead of the texture slice. Viewports reside in the rasterizer stage, and hence have not been discussed yet. However, we can generalize a viewport to represent a subregion within a render target. Multiple viewports can be bound simultaneously, with each representing different portions of a render target. The usefulness of these viewports is that different representations of a scene can be rendered into each viewport. This is commonly done for split-screen rendering, where each side of the screen shows one player's view of the scene. The geometry shader stage's instancing abilities can also be used to provide an index for use as the viewport array index, effectively allowing multiple sets of geometry to be passed to multiple viewports.

## 3.9 Stream Output

In the geometry shader stage, we saw how its stream output object is used to produce a stream of primitives that is passed on to the rasterizer stage, where they are ultimately used to modify the contents of a render target later in the pipeline. However, this is not the only use of the output streams. We have also seen how the stream can be sent to the stream output stage, where the output primitives can be streamed into a buffer resource for later use. This is the sole purpose of the stream output stage—to provide a mechanism for connecting the geometry shader and output buffer resources to hold the vertex data. The location of the stream output stage in the pipeline is highlighted in Figure 3.43.

As mentioned above, it is possible to use more than one stream output buffer at the same time. This allows the geometry shader to produce four different versions of its output, and it provides a significant amount of freedom to implement a wide variety of algorithms. The stream output stage can be considered a fixed-function pipeline stage, although no real processing is performed within the stage itself. Instead, the application configures the stream output stage by using a specially created geometry shader program, and then binds the desired buffer resources to the stage. Since it serves as an exit point from the pipeline, there is no pipeline stage attached to the output of the stream output stage.

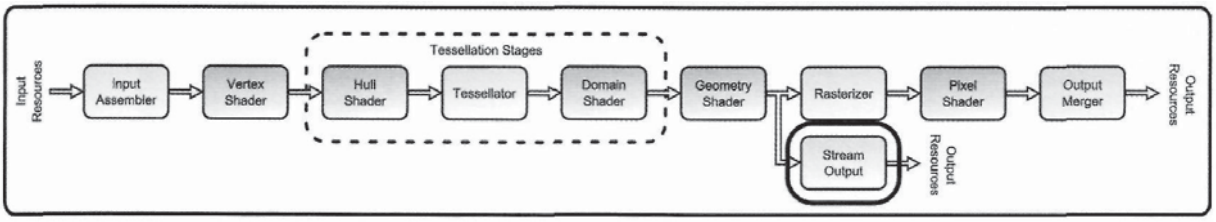



Figure 3.43. The stream output stage.

### 3.9.1 Stream Output Pipeline Input

The stream output stage can only receive information from the geometry shader stage in the form of the output streams declared in the geometry shader program. Up to four streams are available to the geometry shader to pass data to the stream output stage, and each stream can receive a different vertex structure as its input. This allows multiple variations of the same geometry to be streamed simultaneously. For example, the positions, normal vector, tangent and bitangent vectors, or other optional per-vertex data, can be split into multiple streams and then selectively bound to the input assembler stage later on, providing a very flexible method of binding only the vertex data that is required for a particular effect.

The restrictions on these vertex structures are that a maximum of 128 scalars is allowed for each structure, and that a geometry shader invocation may not produce more than 1024 scalars. Thus, if four output streams are used, and each uses a structure of 128 scalar vertices, a total of 512 scalars for all four streams can be written twice. In practice, this should be a sufficient amount of data to stream, since it is performed for each primitive that the geometry shader receives.

It is also possible to produce completely disjoint streams of data as well. For instance, if an algorithm requires that all front faces must be rendered separately from the back faces, then the front and back faces for a set of geometry can be passed to two different streams. This provides a simple mechanism for generating these mixed sets of geometry from a standard geometry format.

### 3.9.2 Stream Output State Configuration

The stream output stage requires two different types of configurations. The first is that the application must bind the appropriate number of buffer resources to this stage, through the ID3DllDeviceContext::SOSetTargets(...) method. This method operates in a similar way to other resource binding functions, and it allows up to four buffers to be bound simultaneously from within the same method invocation. Listing 3.18 demonstrates how this method is used. There is also a corresponding Get method that can be used to retrieve references to the buffer resources that are currently bound to the stage.

```
ID3DllBuffer* pBuffers[l] = { pBuffer1 };
UINT aOffsets[1] = { 0 };

pContext->SOSetTargets( 1, pBuffers, offset );
```

Listing 3.18. Binding buffer resources to the stream output stage.

Each buffer resource that will be bound for output from the stream output stage must be created with the D3D11_BIND_STREAM_OUTPUT bind flag, in addition to any other bind flags that indicate how the buffer will be used for vertex buffer usage. The buffer resource references that are stored in this stage persist between pipeline executions, meaning that they must be managed by the developer. This is the same principle that other resource-binding functionalities use, and the same management techniques can be employed. As we have seen before, it can be beneficial to either always bind all four buffer entries, with the unused slots receiving a NULL pointer, or to have the current state of the slots managed by the application, modifying only the appropriate slots as needed.

The second configuration required for the stream output stage to operate is the specification of what data will be streamed to which output slot. This information is provided to the ID3DllDevice: :CreateGeometryShaderWithStreamOutput(...) method in the form of an array of D3D11_S0_DECLARATION_ENTRY structures. The elements of this structure are specified in Listing 3.19.

```
struct D3D11_S0_DECLARATI0N_ENTRY {
    UINT   Stream;
    LPCSTR SemanticName;
    UINT   Semanticlndex;
    BYTE   StartComponent;
    BYTE   ComponentCount;
    BYTE   OutputSlot;
};
```

Listing 3.19. The members of the D3D11_S0_DECLARATI0N_ENTRY structure.

In this structure, we can see the information required for each piece of every vertex output by the geometry shader that will end up in one of the stream output buffers. One of these structures must be provided for each output attribute that will be streamed out. The first argument, Stream, identifies which stream output object the data will be coming from in the geometry shader. The SemanticName attribute provides the semantic that is defined in the geometry shader for the attribute to be streamed, and the Semanticlndex attribute provides an index for multiple attributes that share the same semantic name to be uniquely identified. This is the same type of semantic index that we have seen for vertex buffer element declarations.

The StartComponent and ComponentCount arguments determine which portion of a four-component attribute will be streamed. StartComponent can have a value of {0,1,2,3}, which corresponds to the register components *{x,y,z,w}* respectively. The ComponentCount simply indicates how many of these components to stream. For example, if StartComponent

is 1 and ComponentCount is 2, the *{y,z}* components would be streamed. The final argument for this structure is OutputSlot, which determines which stream output stage buffer resource will receive the streamed data. The valid values range from 0 to 3, to correspond with as many as four output buffers.

Once an application has properly filled in an array of these structures to define the information to be streamed out, the geometry shader object can be created. Listing 3.20 shows the method used to create this special version of the shader object.

```
HRESULT CreateGeometryShaderWithStreamOutput(
    const void *pShaderBytecode,
    SIZE_T BytecodeLength,
    const D3D11_SO_DECLARATION_ENTRY *pSODeclaration,
    UINT NumEntries,
    const UINT *pBufferStrides;
    UINT NumStrides,
    UINT RasterizedStream,
    ID3D11ClassLinkage *pClassLinkage,
    ID3D11GeometryShader **ppGeometryShader
);
```

Listing 3.20. The CreateGeometryShaderWithStreamOutput() method of the ID3D11Device interface.

We are the most interested in seeing which of these parameters is different than the normal geometry shader object creation method. The first new parameter, pSODeclaration, is the pointer to an array of the declarations described above. This is followed by NumEntries, which indicates how many entries are in the array. The next two parameters, pBufferStrides and NumStrides, provide the vertex strides of each of the vertices that will be streamed to each buffer. This allows each buffer to use a different vertex format to store the desired information. The final new parameter is RasterizedStream, which provides the index of the output stream that should be sent to the rasterizer stage. If none of the streams will be rasterized, this parameter should be set to the constant D3D11_SO_NO_RASTERIZED_STREAM. Whichever stream is specified for rasterization, it can still provide data to one of the stream output buffers as well.

When a geometry shader object is created with this method and then subsequently bound in the geometry shader stage, it activates the stream output stage. Then when a pipeline execution begins, any data passed into an output stream in the geometry shader is matched up with the output configuration based on its semantic name and semantic index. Once a match has been found, that data is streamed to the appropriate buffer to create the completed output vertices.

## 3.9.3 Stream Output Stage Processing

### Automated Drawing

There are two general ways to make use of the stream output functionality. The first is to actually use the resulting buffer resource as an input to the input assembler stage for supplying the vertex data for a pipeline execution. In this case, a special draw call is used to execute the pipeline. The buffer resource must be bound to the input assembler stage slot 0, and the corresponding input layout object must also be configured by the application. Then the ID3DllDeviceContext::DrawAuto() method will inspect the contents of a streamed-out buffer that was bound to the input assembler stage. It then provides the appropriate number of vertices and primitives to the input assembler to render the entire contents of the buffer, with no interaction from the application. In this case, the geometry shader object produces the streamed output vertices[23] and the primitive connectivity information is determined by the input assembler's primitive type configuration. The primitive type should be chosen to be compatible with the type of primitive stream used to stream the data out. If it is then rendered with the DrawAuto method, the complete rendering is performed without the application having knowledge of the contents of the buffer. This is another way in which GPUs are becoming more autonomous and can operate somewhat independently of the CPU.

The utility of this operation may not be immediately obvious. After all, if the geometry reaches the geometry shader, we could just as easily pass its output stream to the rasterizer stage and use it directly—what benefit is there to streaming a mesh out and then re-rendering it? The important point to consider is that the complete set of operations that have been performed between the input assembler and the geometry shader are included in the output stream generated by the geometry shader. Any vertex transformations performed in the vertex shader can be saved, allowing any further rendering passes requiring the same geometry to skip the vertex transformations and continue directly to the next stage. When an expensive vertex-level operation is performed, such as vertex skinning, this can result in a significant reduction in the number of calculations performed during a rendering pass.

This is also especially useful for algorithms that require multiple rendering passes when tessellation has been used. If a model's geometry is first transformed in the vertex shader, and then tessellated in the tessellation stages, and finally streamed out in the geometry shader stage, the results are available for future rendering passes without having to perform all of those calculations again. The tessellation stages can be quite expensive to compute if a very high tessellation level is used, which means that caching the tessellated mesh in this way can save a significant number of calculations. The processing savings are depicted in Figure 3.44.

[23] If either a triangle stream or a line stream is used to produce the output vertex data, the strips that are passed into these streams are converted into lists of primitives. This provides a consistent data format for use with the DrawAuto() method.

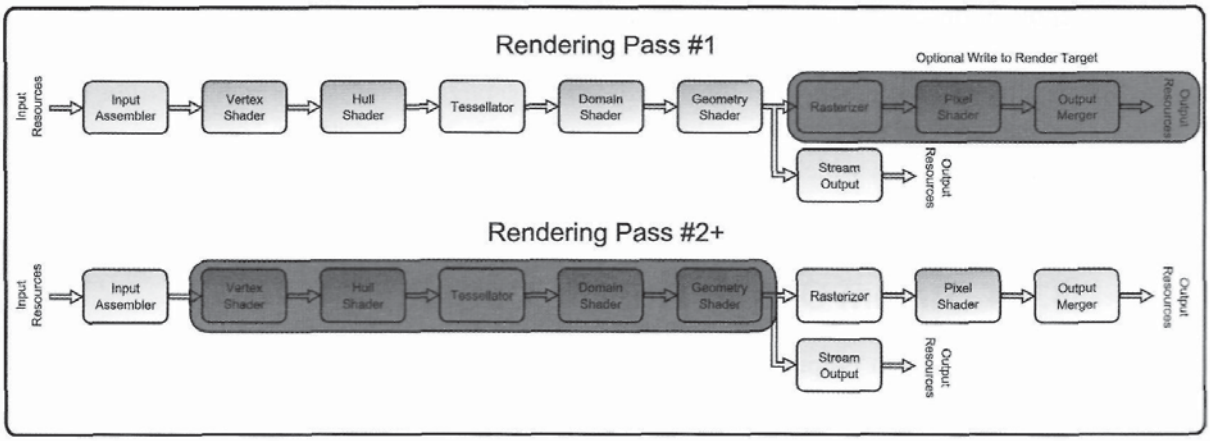



Figure 3.44. The benefits of caching tessellated geometry for multiple rendering passes.

## Stream Output as a Debugging Tool

Another potential use for the stream output stage is to tap into the pipeline data at the geometry shader stage and retrieve a portion of the data for debugging purposes. This can be helpful for inspecting intermediate values that would normally not be available in the final rendered output. In this case, two different buffer resources are needed. The first is the stream output buffer, which must be created with the default resource usage flag. The application cannot access this type of buffer directly, since it is not possible to create buffers that can be written to by the pipeline and also read from the CPU. Instead, we create a second buffer with the staging usage flag, and then copy the streamed data from the default usage buffer to the staging usage buffer. The application can then map the staging buffer and read its contents.

Once the contents are accessible to the application, they can be accessed by using a structure that identifies the layout of the streamed vertices. This is typically known well in advance, since the stream output configurations are needed before executing the pipeline, anyway. The number of primitives written to the buffers can be retrieved with a pipeline query object of the type D3D11_QUERY_DATA_SO_STATISTICS (more details about how to perform a pipeline query can be found in Appendix B). This allows the application to know at what point in the buffer the pipeline-generated data ends, and where the default or invalid contents of the buffer begin.

With the buffer data accessible in an evaluated form, the application can use the data in whatever way is needed. This can include writing the data to a file for offline inspection, analyzing data, such as finding the axis-aligned bounding box of the new data, or even printing statistics to the screen in real time for a live visualization of the model data being passed through the geometry shader. These types of techniques would typically not be used for a user-facing application, but they can provide a number of debugging options during

application development. The other main debugging tool for Direct3D 11 applications, the PIX tool, is used primarily for offline analysis of data. This makes streaming of pipeline data an appropriate method for providing live debugging information. In addition, the use of the stream output functionality allows for saving intermediate values from within the geometry shader, which are typically not available for reading in PIX.

### 3.9.4 Stream Output Pipeline Output

Since the stream output stage doesn't have another stage attached to its output, we can consider its output as consisting of the data streamed to the buffers. This is just one of several ways to extract data from the pipeline, but depending on the configuration of the pipeline, it can be used as the primary output. For example, the pipeline could be used as a transformation and tessellation accelerator for a software-based rendering system. In such a scenario, the geometry that will be rendered would be transformed and tessellated by the GPU and then would be streamed out and read back to the CPU, where it could be used in the software renderer. The geometry would never be rasterized by Direct3D 11, but could instead be rendered with a ray-tracing engine or some other high-quality techniques not available on the GPU.

## 3.10 Rasterizer

The geometry shader stage marks the final point in the pipeline that deals strictly with geometric data. Regardless of whether it receives data directly from the vertex shader stage, the domain shader stage, or the geometry shader stage, the geometric pipeline data is fed into the rasterizer stage. The location of the rasterizer stage is highlighted in Figure 3.45.

The primary purpose of this stage is to convert the geometric data into a regularly sampled representation that can later be applied to a render target. This sampling process

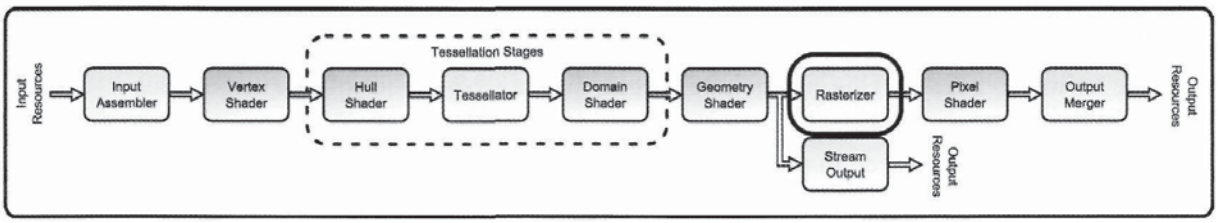



Figure 3.45. The rasterizer stage.

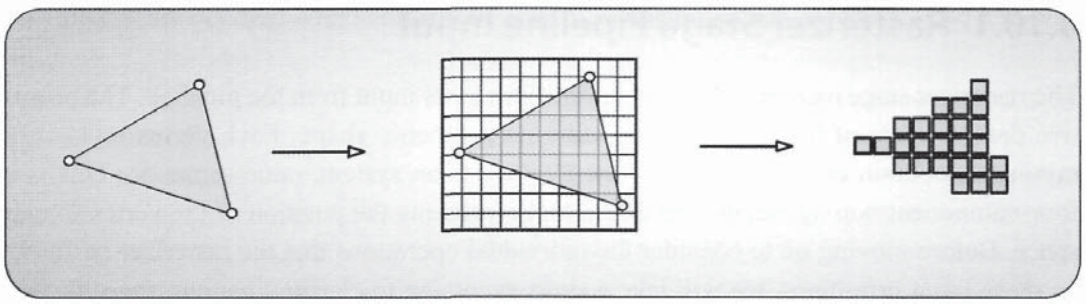


Figure 3.46. A triangle that has been rasterized.

is referred to as *rasterization,* and it maps individual primitives to a format appropriate for storage in a texture resource. The result of rasterization is to produce a number of *fragments* that approximate the original geometry. This is the first step toward generating individual pixel data that will contribute to an output image. Figure 3.46 shows what a rasterized triangle looks like.

In addition to performing the rasterization operation, the rasterizer stage also performs a number of additional operations before fragments are generated. The first operation is *primitive culling,* which eliminates primitives that will not contribute to the output rendering due to their location within clip space. By eliminating primitives prior to rasterization, the overall number of primitives to process is reduced, which increases the efficiency of the stage.

The second operation is the clipping of primitives. This process "clips" primitives to the portion of clip space that can affect the rendered output. This essentially cuts up primitives that are partially in the viewable area into primitives that are completely within it. The portion outside of this region is not needed since it won't produce fragments that can affect the output render target. The clipped primitives are then mapped to the active viewport. The viewport is a structure that describes a subset of the current render target to receive the processed primitives. Finally, the *scissor test* is applied during the rasterization process. This essentially allows the application to specify a rectangular region where rasterization is allowed to occur. Any generated fragments that fall outside of the scissor rectangle are discarded instead of being passed further down the pipeline. This also improves overall pipeline efficiency when only a subregion of the rasterized output will be used.

With all of these operations to perform, the rasterizer stage contributes a large amount of functionality to the processing of pipeline data. It can have a profound impact on the performance of the pipeline as a whole, and hence it is important to understand how all of the processes mentioned above work, and how to influence their operation. The output from the rasterizer stage sets up the rest of the pixel-based pipeline stages, which makes this a data amplification point.

## 3.10.1 Rasterizer Stage Pipeline Input

The rasterizer stage receives individual primitives as its input from the pipeline. The primitive data consists of the vertices that define its geometric shape. Each vertex must, at a minimum, contain one attribute with the SV_Position system value semantic. This is a four-component homogeneous coordinate that represents the position of the vertex in clip space. Before moving on to consider the individual operations that the rasterizer performs on these input primitives, we will take a short excursion to clarify what clip space is, and how geometry is transformed into this coordinate space. Perhaps more importantly, we will consider the properties of this coordinate space, and how it is used in the context of rasterizing a scene.

### Clip Space and Normalized Device Coordinates

The term *clip space* refers to a post-projection coordinate space that can be used to identify what portion of a scene will be visible under the current viewing conditions. This space is essentially a result of the current projection transformation, which is typically either a perspective or orthographic projection. We are primarily concerned with perspective projections, and will focus on them throughout this section. However, many of the same concepts apply to orthographic projections as well, so the following discussion is valid in both cases.[24]

We mentioned above that the position data supplied in the SV_Position system value semantic are homogeneous coordinates consisting of four components. In general, the first three components of this position specify the *X*, *Y*, and Z coordinates of the vertex in the 3D space in which it currently resides. The fourth component, which is referred to as the *W* coordinate, always has a value of 1 before a projection transformation is applied to it. Thus, the position coordinates are always in a form such as *[X,Y,Z,\]*. After a projection transform is applied to these points, the resulting form will depend on the type of projection that was performed. If a perspective projection is used, the resulting point produces a *W*-value that is equal to the Z-component prior to the projection[25] while the Z component is a scaled version of itself, based on the projection transformations' near and far clipping planes. Equations (3.2) and (3.3) show the results of the projection transformation on an input Z (and *W)* component based on the projection matrix shown in Equation (8.7). Here we assume that the input coordinates are denoted by the subscript 1, and that the post-projection

[24] A general discussion of the various transformations that a set of geometry pass through is provided in Chapter 8, "Mesh Rendering." This includes a discussion of the perspective projection.

[25] This assumes that the projection matrix used is the same as described in the DXSDK documentation, and also as described in Chapter 8.

coordinates are denoted with subscript *p*. The following equations are used:

$$z_p = \frac{z_f(z_1 - z_n)}{(z_f - z_n)}; \tag{3.2}$$

$$w_p = z_1. \tag{3.3}$$

The result of Equation (3.2) may not be obvious at first inspection, but we can gain some insight into this equation by plotting z after the divide by *W* as a. function of the input Z-coordinate. This is demonstrated in Figure 3.47. Here we can see that the post projection Z-value can reside in one of three general areas. If the input Z-coordinate is less than the near clipping plane, $Z_n$, the resulting $Z_p$ has a negative value. If the input Z-coordinate is between $Z_n$ and $Z_f$ the resulting $Z_p$ has a value between 0 and $Z_f$ Any input Z-coordinate that is greater than $Z_f$ produces a $Z_p$ that is greater than $Z_f$[26]

At some point after projection, these post-projection coordinates must be rehomogenized by dividing by the *W*-coordinate. By dividing all of the coordinates by the *W*-coordinate, we ensure that the post-divide *W*-coordinate will be 1. We can also consider what happens to our post divide Z-coordinate by simply dividing the results discussed above by $Z_1$. When $Z_1$ equals $Z_n$, the result is 0. When $Z_1$ equals $Z_f$, the result is 1. Thus, after the divide by *W* we have a valid depth range of values between 0 and 1, which correspond to the input values between $Z_n$ and $Z_f$ We won't repeat the calculations here, but the *X*- and *Y*-coordinates produce a similar behavior, with each of them producing valid values in the -1 to 1

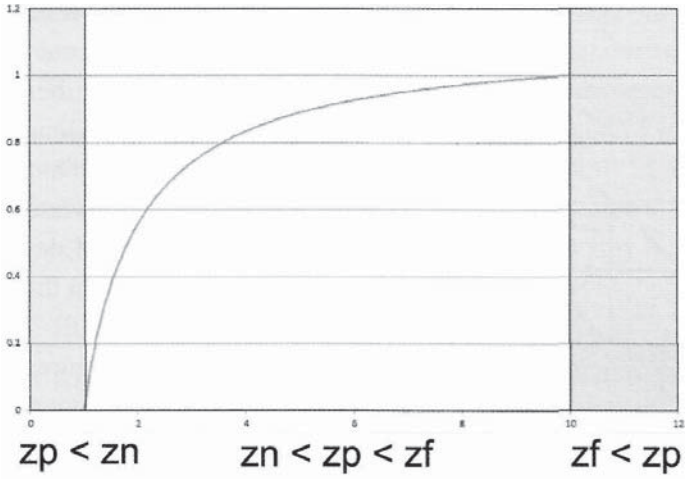



Figure 3.47. The post-projection Z-coordinate of a position.

[26] All of these equations assume a left-handed coordinate system, where Z is positive in the direction of view. This is the standard coordinate system used in Direct3D.

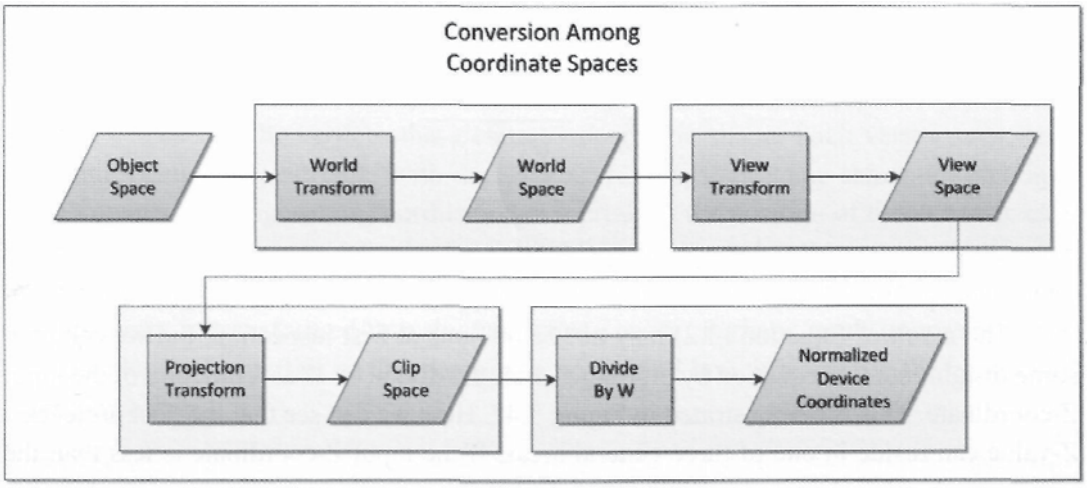



Figure 3.48. The change from one coordinate space to another.

range in the post-*W*-divide range. Thus, we have two different ways to express the post-projection coordinates—before and after the *W*-divide. We will refer to the pre-divide coordinates as residing in clip space, and we will refer to the post-divide coordinates simply as normalized device coordinates. This distinction is very important, since both coordinates describe the same point, but their values will vary significantly between the two. The various coordinate spaces and the operations that produce them are shown in Figure 3.48.

As we noted above, the position received by the rasterizer stage in the SV_Position system value semantic should be the result of the projection transformation, and hence should be in clip space. The rasterizer stage performs the *W*-divide later in its sequence of processing, so the shader program that performs the projection transformation doesn't need to manually execute the division.[27] In many cases, it can often be easier to visualize the post-projection coordinates in terms of the normalized device coordinates when considering whether a particular vertex will be visible or not. In these normalized device coordinates, a vertex must reside within the cube between the points [1,1,1] and [-1,-1,0]. This range of coordinates is depicted in Figure 3.49.

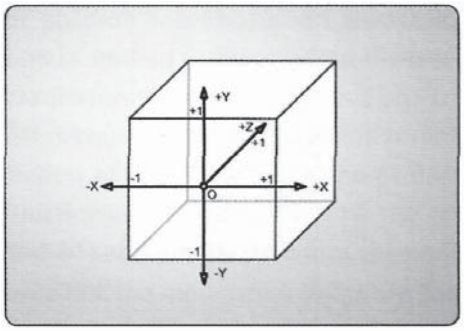



Figure 3.49. A visualization of the region where vertices are within the current viewing area.

As we can see in Figure 3.49, the positive X-axis points to the right, the positive 7-axis points upward, and the positive Z-axis points

[27] This doesn't mean that the shader can't perform the W-divide if it needs to for further calculations. If so, the rasterizer stage will simply divide the other coordinates by 1, which will of course produce the same result.

into the scene. This region is often referred to as the *unit cube,* although in reality it is not a complete unit cube, due to its range in the Z direction, but it is still commonly referred to this way. We will explore clip space and normalized device coordinates further in the "Rasterizer Stage Processing" section.

### Clip and Cull Distances

Now that we have an understanding of how the position data received by the rasterizer should look, we can consider some of the other inputs it may receive. The next two input attributes we will look at were initially discussed in the vertex shader section. The system value attributes, SV_ClipDistance and SV_CullDistance are both consumed in the rasterizer stage as additional user-defined inputs to the culling and clipping operations. These allow the culling and clipping processes to be manipulated with customized data provided by the programmable shader programs. We will discuss these in more detail later in the "Rasterizer Stage Processing" section.

### Viewport Array index

In addition to the clip and cull distances, the rasterizer stage can also receive the SV_ViewportArrayIndex system value semantic. This is an unsigned integer that specifies which viewport definition structure the rasterizer should use when rasterizing a primitive. The mechanics of how these individual viewports affect the output will be described in detail in the "Rasterizer Stage Processing" section.

### Render Target Array **index**

We also have seen that the geometry shader can specify the SV_RenderTargetArrayIndex system value semantic to indicate which texture slice of a render target array to write to. This is simply an unsigned integer value that selects the texture slice to target. We don't implement any sample programs that use this system value semantic in the book, but there is a very good Direct3D 10 sample in the DXSDK that demonstrates this technique being used, called CubeMapGS. In addition, a further description of how this is used can be found in (Zink).

### Additional Input **Attributes**

The final group of inputs that can be sent to the rasterizer stage is actually a wide variety of different types of information. Any additional attributes in the vertex structure that are received by the rasterizer represent data that the developer has defined at each vertex of the input primitive. In the case of a triangle, there are three instances of an attribute available for each triangle at each of its vertices. Since the rasterization process generates fragments

that represent a regular sampling pattern between these vertices, the three vertex attribute values must be interpolated to find an appropriate intermediate value to pass on to each fragment. This is performed in the actual rasterization process, and will be discussed later in this section.

## 3.10.2 Rasterizer Stage State Configuration

With such a large number of different functionalities available in the rasterizer stage, there are a correspondingly large number of states to configure in order to control its operation. These configurations are controlled by the application through three different sets of methods, which can be used either to query the existing state or to overwrite the existing state with a new one.

### Rasterizer State

The primary configuration of the rasterizer stage is performed with the rasterizer state object. This object, represented by the ID3D11RasterizerState interface, is an immutable state object that is validated when it is created. As is the case with all objects in Direct3D 11, the object is created through a device method, the ID3D11Device: :CreateRasterizerState(...) method in this case. Like other state objects, this method takes a pointer to a description structure that provides the desired configuration values. Listing 3.21 provides a description declaration, followed by the creation of a rasterizer state object. This is followed by reading the existing state in the current device context, and then setting the new state that we just created.

```
D3D11_RASTERIZER_STATE rs;

rs.FillMode = D3D11_FILL_S0LID;
rs.CullMode = D3D11_CULL_BACK;
rs.FrontCounterClockwise = false;
rs.DepthBias = 0;
rs.SlopeScaledDepthBias = 0.0f;
rs.DepthBiasClamp = 0.0f;
rs.DepthClipEnable = true;
rs.ScissorEnable = false;
rs.MultisampleEnable = false;
rs.AntialiasedLineEnable = false;

ID3DllRasterizerState* pState = 0;

HRESULT hr = m_pDevice->CreateRasterizerState( &rs, &pState );
```

Listing 3.21. The specification and creation of a rasterizer state object.

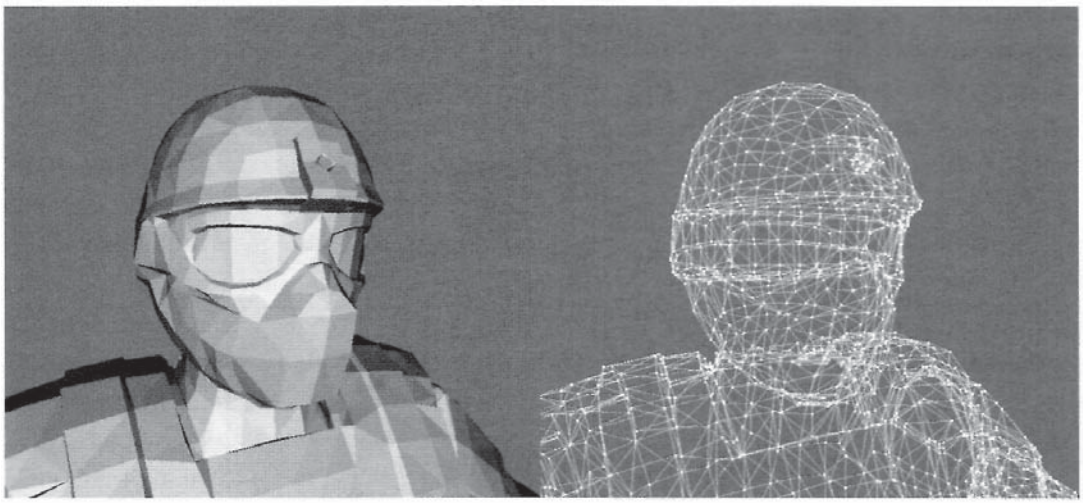


Figure 3.50. The difference between wireframe and solid fill modes. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

We will examine each of the elements of the rasterizer state individually. The first setting is the FillMode parameter, which determines how primitives are rasterized. The available options are `solid` fill or `wireframe`. In solid fill, fragments are generated between the vertices of the primitive to completely fill its interior, while in wireframe mode only the edges of a primitive are rasterized. In practice, this only affects triangle primitives since lines only consist of a single edge, and points only consist of a single point, so the fill modes are equivalent for them. Figure 3.50 shows a simple example of the difference between these two modes for a triangle.

The second setting in this structure is the CullMode. The cull mode controls the cull operation of the rasterizer stage. This can be used to enable culling of primitives that are facing the viewer (*frontfacing)* or facing away from the viewer (*backfacing).* An option is also provided to disable the culling operation entirely. Determining if a triangle is front facing or back facing is done by examining the order that its vertices arrive in. If the vertices are ordered such that traversing them in order produces a clockwise trip around the triangle on the render target, it has *clockwise* vertex ordering. Otherwise, the triangle is said to have *counter-clockwise* winding. The FrontCounterClockwise setting determines if the front-facing triangles should be considered clockwise or counter-clockwise. In effect, this setting is used with the CullMode to select the target of the culling operation.

The next three settings control an optional feature to provide a depth bias to the fragments generated by the rasterizer. Some algorithms require an object to be rendered in two different ways to two separate render targets to achieve a particular effect. *Shadow mapping* is probably the most common example of this, where an object is rendered from a light source's perspective to generate a map of what objects in the scene are visible to the light. This rendering produces what is called a *depth map* for the light, as it determines

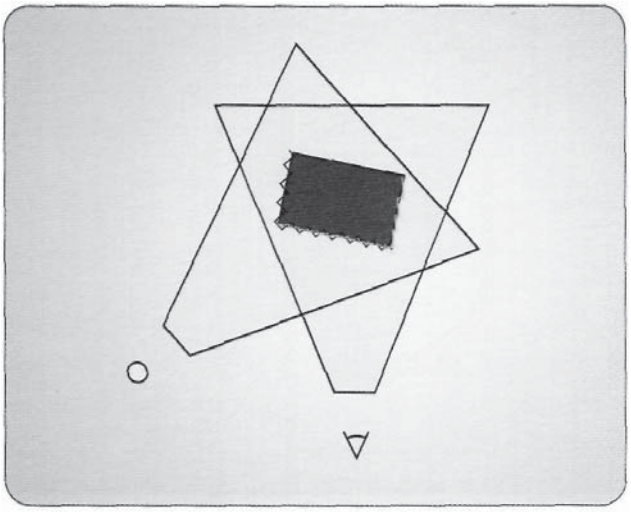


Figure 3.51, A mismatch between a scene rasterized from the light point of view and from the camera point of view.

the distance from the light in each of its pixels. In the subsequent rendering pass, an object is rendered from the viewer's perspective and the depth map is sampled from the first to determine if it can be "seen" by the light and hence would receive light from it. In theory, this is a very sensible way to determine which objects are in shadow and which are not. But in reality, this method can introduce many artifacts, due to differences in the effective sampling patterns of the two rendering passes.[28] This is demonstrated in Figure 3.51, which displays an overhead view of an object and how it is rasterized by both the light's depth map and by the scene rendering.

To combat this type of effect, the rasterizer stage provides a method to introduce a depth bias into the generated fragments. The DepthBias and SlopeScaledDepthBias settings provide a constant and a slope-based offset depth value, respectively. These two parameters apply a depth bias in one of two ways, depending on the type of depth buffer being used. Equation (3.4) and Equation (3.5) provide the two depth calculations.

$$\text{Bias} = (\text{float})\text{DepthBias} * r + \text{SlopeScaledDepthBias} * \text{MaxDepthSlope}; \tag{3.4}$$

$$\text{Bias} = (\text{float})\text{DepthBias} * 2\ (\text{exponent}(\max z \text{ in primitive}) - r) + \text{SlopeScaledDepthBias} * \text{MaxDepthSlope}; \tag{3.5}$$

[28] Many, many, many algorithms have been devised to improve the produced image quality when using shadow maps. A search in academic literature will easily return more than 100 different papers on the topic.

The first technique for calculating the depth bias is used when a depth buffer with a unorm format is used, or if no depth buffer is bound to the pipeline. The second method is used when a floating point depth buffer is used. In the first equation, the r parameter represents the minimum value that can be represented that is greater than 0, and the MaxDepthSlope is the greatest slope in depth found over the primitive. In the second equation, the *r* parameter represents the number of mantissa bits in the floating point type of the depth buffer. In either case, this bias value is calculated and then clamped to the DepthBiasClamp parameter of the rasterizer state description. Finally, the bias value is added to the z value of a vertex after clipping and before interpolation setup is performed. The effect of adding a bias can effectively shift the rasterized surface to ensure that small differences in calculation can't produce artifacts.

The remaining four settings of the rasterizer state description are all Boolean parameters that switch various functionalities on or off. We will explore each of these parameters in more detail in the "Rasterizer Stage Processing" section, but we make a brief mention of them here. The first of these settings is the DepthClipEnable parameter. This enables or disables clipping of primitives, based on their depth values. This essentially clips the geometry to the near and far clipping planes. Next is the ScissorEnable parameter, which enables or disables the use of the scissor test. The scissor test is used to cull any fragments that are generated outside of the scissor rectangle. The third Boolean parameter is multisampleEnable, which essentially toggles whether the rasterizer performs multiple coverage tests for MSAA render targets. And finally, AntialiasedLineEnable enables or disables the use of anti-aliasing when a line primitive is rasterized.

## Viewport State

In addition to the main rasterizer state, the application must also provide at least one viewport for use in rendering operations. The viewport structure consists of six floating point parameters that identify the subregion of the current render target that will be rasterized into. Listing 3.22 shows the contents of this structure.

```
struct D3D11_VIEWP0RT {
    FLOAT TopLeftX;
    FLOAT TopLeftY;
    FLOAT Width;
    FLOAT Height;
    FLOAT MinDepth;
    FLOAT MaxDepth;
}
```

Listing 3.22. The members of the D3D11 VIEWPORT structure.

As seen in Listing 3.22, the viewport structure defines a region that the normalized device coordinates will be mapped to in order to convert from the unit-sized coordinates of the unit cube to the pixel based coordinate system of a render target. As an example, if the entire render target should be covered by the viewport, the top and left values will be 0 and the width and height parameters will be the width and height of the render target being used. The MinDepth and MaxDepth parameters are used to scale the range of depth values being used to a subset of the complete regular range.

The application can either provide a single viewport, or it can specify multiple viewports simultaneously. If multiple viewports have been set, the viewport that is used for a particular primitive rasterization is determined by the value of the SV_ViewportArrayIndex system value semantic. Binding of viewports is performed with the ID3DllDeviceContext:: RSSetViewports method. Unlike the state arrays that we have discussed previously, any viewports that are not set in the most recent set call will be cleared. In addition, there is the usual Get counterpart method for retrieving the currently set array of viewports.

#### Scissor Rectangle State

The final configuration available in the rasterizer stage is the ability to specify an array of scissor rectangles. These rectangles are used during the scissor test to specify a particular region of the render target for which fragments can be generated. This effectively eliminates any fragments that would have been generated for a region outside of the current scissor rectangle, which can reduce unnecessary computation. If multiple scissor rectangles are bound, the rectangle used for a given primitive is determined by the same SV_ ViewportArraylndex. This ensures that the viewport and scissor rectangle that are used together are a matching pair. The array of scissor rectangles is also managed in the same way as the array of viewports. It can be bound using the ID3DllDeviceContext: :RSSet ScissorRects() method, making only the scissor rectangles that were specified in the most recent set call remain active.

### 3.10.3 Rasterizer Stage Processing

All of the processes performed in the rasterizer stage are vital components for efficiently converting geometric data into image-based data suitable for storage in a render target. This section explores in greater detail what each of these processes encompasses, and attempts to provide some insight into how they can be used in common real-time rendering contexts. Figure 3.52 shows a block diagram of the functional operations performed within the rasterizer stage. These processes may be implemented in a different order in the actual GPU hardware, but we will conceptually consider them in the shown sequence.

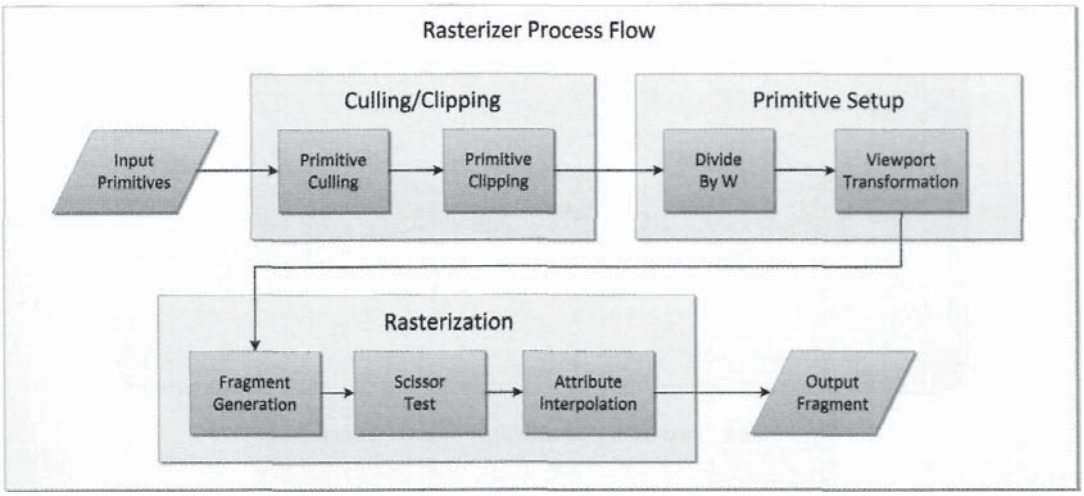



Figure 3,52. The functional operations performed by the rasterizer stage.

As can be seen in Figure 3.52, the rasterizer receives complete, individual primitives as its input. At this point in the pipeline, any primitive topologies with adjacency are converted to standard primitive topologies. If the primitives were passed into the pipeline as strips, those strips are broken apart and issued to the rasterizer stage individually. These primitives are then processed in a sequential pipeline fashion, with the process ending with the production of fragments, which are then passed to the pixel shader stage. The following sections explore in more detail what is performed in each of these functional blocks.

## Culling

The first operation performed on the incoming primitives is culling. This operation is intended to remove primitives that will not contribute to the final rendered image from further processing, to increase the efficiency of the pipeline. There are two different types of culling: *back face culling* and what we will refer to as *primitive culling.*

**Back face culling.** The first form of culling we will examine is back face culling. As its name indicates, this operation is only applied to triangle primitives, since they are the only basic primitive with a concept of a face. However, triangles are typically the most frequently used primitive, and hence this process is quite important to most rendering sequences. When the vertices of the input primitive are received by the rasterizer stage, the winding of the vertices is determined. Figure 3.53 shows two different triangles, where the vertex winding is either clockwise or counter-clockwise. One technique for performing this check is described in the "Vertex Shader Pipeline Output" section, but the hardware implementation may vary.

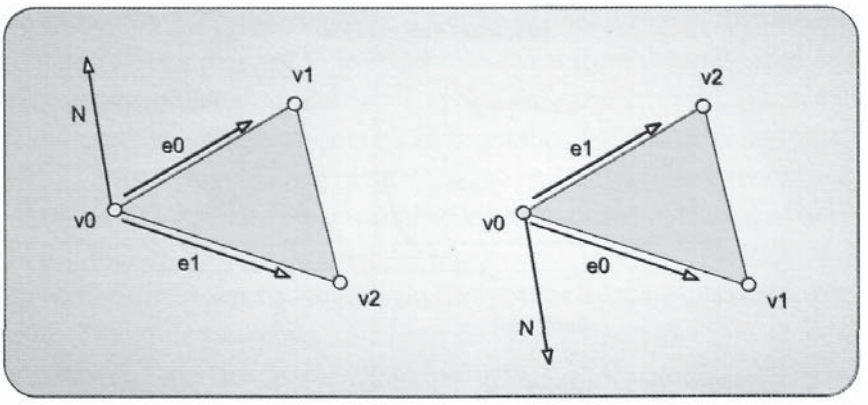



Figure 3.53. Two triangles with differing vertex windings.

We have seen that in the rasterizer state configuration, there are two different settings for the culling operation: CullMode and FrontCounterClockwise. Specifying which vertex winding represents a front-facing primitive depends on the input data introduced to the pipeline in the input assembler stage. The digital content creation tool that was used to generate the input geometry may use either winding, but it is common to have the option to reverse the winding, to make the model compatible with the convention of the end user's application. Reversing the vertex winding order is performed by swapping two of the vertices from their current locations in the primitive's list of vertices.

Once the convention of what represents the front face of a primitive has been specified by the FrontCounterClockwise parameter, the CullMode parameter determines which faces, if any, should be culled. Culling is used to reduce the amount of work required by the rasterizer stage by eliminating triangles that are facing away from the current view point. If a triangle faces away from the current view, it is considered to be back-facing. In traditional opaque rendering algorithms, back faces do not contribute to the final image and should be eliminated. In fact, roughly half of the primitives that enter this stage will be removed, since only half of a model can be visible in a single view within a frame. In this case, the CullMode parameter should be set to D3D11_CULL_BACK.

At the same time, there are many algorithms that require rendering geometry multiple times with different cull modes to achieve different effects. For example, a simple technique for finding the thickness of an object at every pixel is to first render it with front faces being culled and writing to the red channel of the render target, while using a blending state that selects the maximum value when writing to the render target.[29] A second rendering pass is performed with back faces being culled and writing to the green channel of the render target, while using a blending state that selects the minimum value when writing to the render target. After these two passes, the render target contains the farthest point in the red

[29] Blending will be covered in more detail in the output merger section of this chapter.

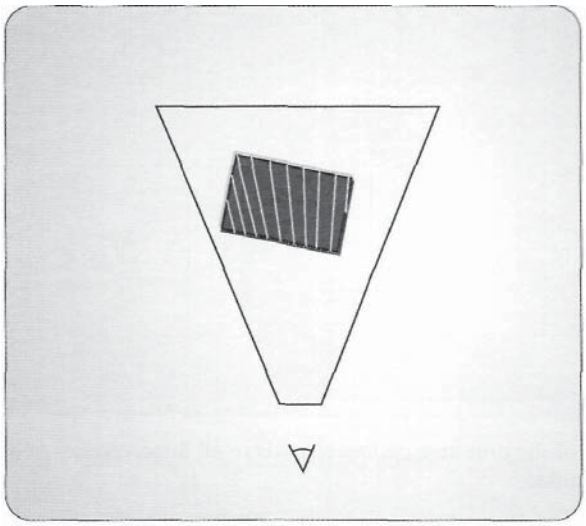


Figure 3.54. A depiction of the minimum and maximum depths of an object, enabled by using varying culling modes.

channel, and the nearest point in the green channel. Together, the difference between these two values provides an estimate of the thickness of an object.[30] This concept is depicted in Figure 3.54. In algorithms such as this, it is necessary and useful to cull triangles with one orientation in one pass, and then cull the opposite orientation in the second pass. A number of special effects can be achieved by manipulating the culling order.

**Primitive culling.** The second form of culling that is performed in the rasterizer stage tests if there are any primitives that reside completely outside of the unit cube in normalized device coordinates. This can be conservatively determined by testing if all of the vertices of a primitive are outside of the same clipping plane, where the clipping planes are defined by the faces of the unit cube. This operation can be performed very efficiently when performed in clip space,[31] since the planes are axis-aligned and are always located at the same distance from the origin. For example, to test if the three vertices of a triangle are all outside of the top plane of the unit cube, the test simply consists of checking if the F-component of their clip space positions is greater than its *W*-component. The test can be performed by taking the difference of the *W*-component minus the *Y*-component. If all of the results are negative, the primitive can be safely discarded from further processing. This

[50] There are other, newer techniques that can perform this depth calculation in a single pass, but the example is still a valid situation where multiple rendering passes are required with switched culling order.

[31] Which space this is performed in is up to the hardware implementation, and may or may not be performed in clip space.

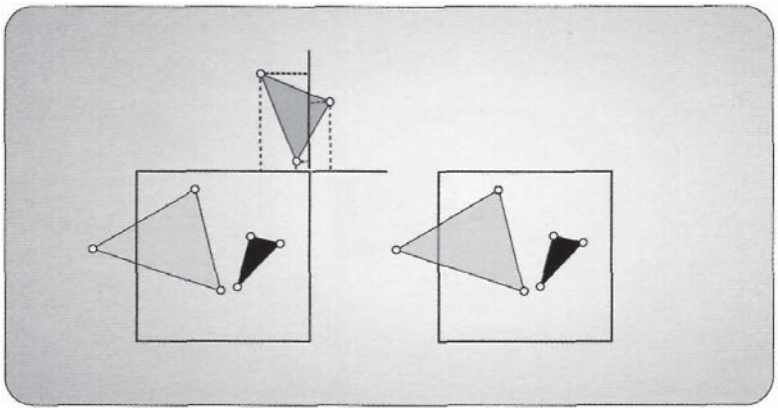


Figure 3.55. A depiction of the primitive culling test where all three vertices of a triangle must be outside of the same plane to be culled.

check is performed for each of the sides of the unit cube. This culling process is depicted in Figure 3.55.

In addition to this standard form of culling, the developer can also use custom culling algorithms. We have seen the system value attribute SV_CullDistance in the vertex shader section. This system value semantic will cull a primitive if all of its vertices arrive at the rasterizer stage with negative values in the same attribute component marked with the SV_CullDistance semantic. The value can be generated with a user-defined plane equation, performing a distance test and then storing the scalar result in the SV_CullDistance attribute. In practice, there is no limitation imposed on the type of calculation used to generate the culling value. If there is a more appropriate test to be applied, it can be implemented as needed and the result stored in the same way. For example, when generating a paraboloid environment map, geometry could be culled that exists in the opposite hemisphere from the one being generated at the moment (paraboloid maps are discussed in more detail in Chapter 13).

### Primitive Clipping

The next operation performed by the rasterizer stage is primitive clipping. This takes each primitive that has survived the culling operations and tests if it is located fully inside of the unit cube, or if it is only partially inside. If the primitive resides completely outside of the unit cube, it can be discarded outright without any further processing. If it is partially inside, then the primitive is split into new primitives that reside entirely within the unit cube, and the exterior primitives are discarded. This procedure is depicted in Figure 3.56. Since the unit cube corresponds to the view frustum of the current projection matrix, this operation is also referred to as frustum clipping.

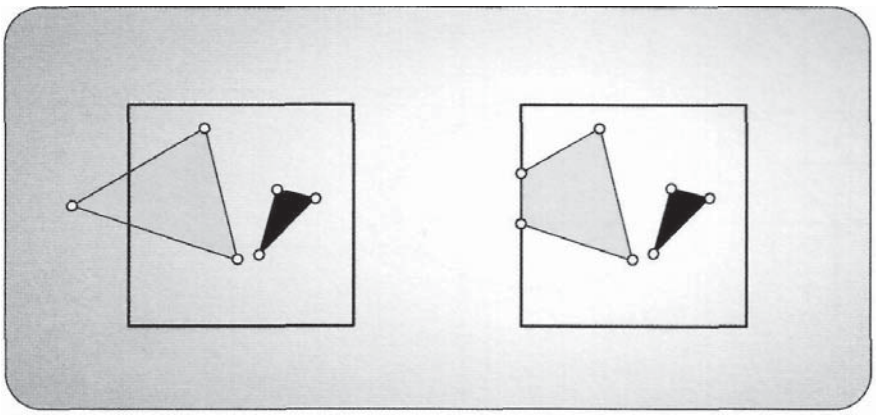


Figure 3.56. A depiction of the clipping process.

The implementation of this process is hardware dependent, but it will generate new vertices at the boundary of the unit cube, to allow the new primitives to be created. An important point to consider during this process is that these generated vertices receive interpolated attribute values from the original vertices, which will then be used later to perform the actual rasterization process.

Like the culling mechanism, the primitive clipping hardware can also be used by the developer to perform customized per-fragment clipping. By using the SV_ClipDistance system value attribute, each vertex can specify its location relative to a user-defined clipping function. This can be used to implement custom clipping planes, or some other function that will provide an appropriate clipping result. These attributes are interpolated across the primitive, and a per-fragment clipping test is performed. If the interpolated attribute value is negative, then the fragment is discarded. Multiple clipping functions can be implemented by using more than one component of a vertex attribute. For example, if a vertex attribute is declared as a float4 type, then the results of the four different clipping functions can be specified with one result in each component. This allows for a significant number of tests to be performed simultaneously.

### Homogenous Divide

Once a primitive has passed through the culling and clipping operations, the *homogenous divide* is performed. This process simply divides the projected points by their *W*-components, as we discussed earlier, producing homogenous coordinates of the form *[X/W, Y/W, Z/W,* 1]. This format is referred to as *normalized device coordinates,* since the size of each coordinate has been scaled by the *W*-coordinate. Prior to this conversion, the coordinates are still in clip space, which is the form produced by the projection matrix. The input clip space points may or may not have a *W*-coordinate value of 1.

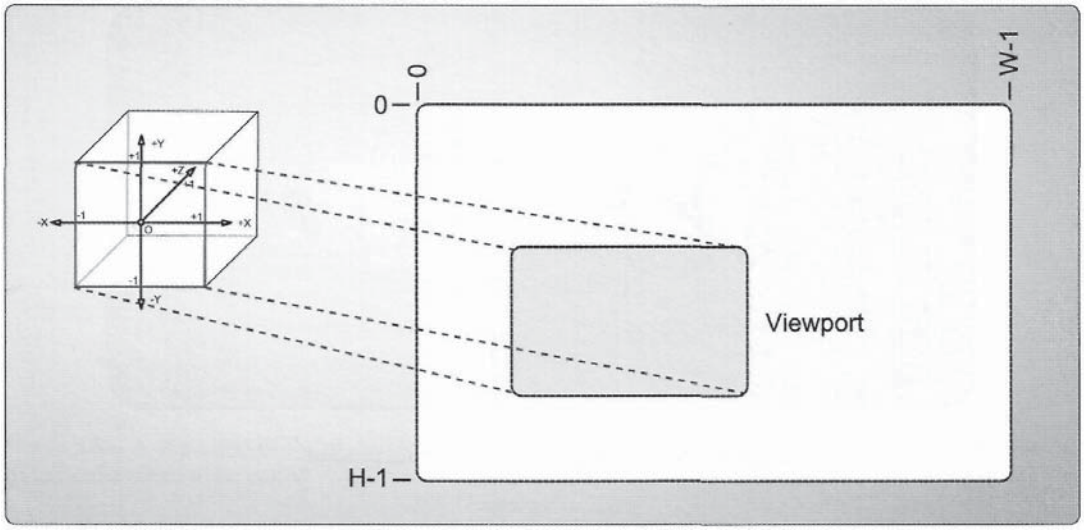



Figure 3.57. The mapping of the unit cube in normalized device coordinates to a render target's screen space coordinate system.

## Viewport Transformation

The final step before the actual rasterization takes place is the *viewport transformation*. The vertices of the primitive currently reside in normalized device coordinates, where the *X*- and *Y*-coordinates range between [-1,1] and the Z coordinate ranges between [0,1 ]. The viewport provides the information needed to map the primitive from normalized device coordinates to pixel coordinates. The viewport data structure provides an offset (with the TopLeftX and TopLeftY parameters) and a scale (with the Width and Height parameters) for positioning the primitives within the desired region of a render target. This mapping process is depicted in Figure 3.57.

After the mapping has been performed, the new coordinates will have a range of [TopLeftX,TopLeftX+Width] for the *X* component, and [TopLeftY,TopLeftY+Height] for the *Y* component, where width and height are the dimensions of the render target area to be rendered to. It is also possible to scale the Z-component of the position as well, to manipulate the depth values used later in the pipeline. As we have mentioned previously, multiple viewports can be used to achieve effects such as split screen rendering, where the scene is rasterized into two different regions of a render target to indicate what is visible from each player's current view point. However, this could also be used to only render a scene to a region that isn't covered by a user interface element. For example, it is common in real-time strategy (RTS) games to have a large user interface element at the bottom of the screen. If this user interface is not transparent, then it can significantly reduce the size of the region being rasterized into, and subsequently reduce the number of fragments/pixels that need to be generated. In general, if there is a reason to restrict the region that is being rendered with a viewport, it is a good practice to do so!

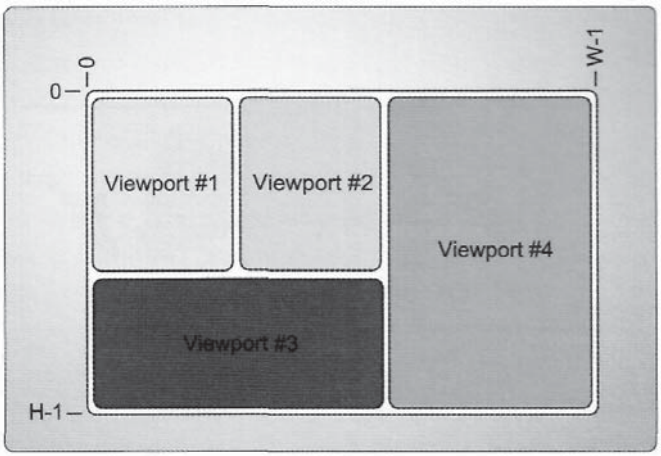



Figure 3.58. Multiple viewports within a single render target, which allows rasterizing of geometry to a particular region of the render target.

There are two different ways that the desired viewport can be selected by the application. If the input signature to the rasterizer stage (as determined by the declared output from the previous stage) contains the SV_ViewportArrayIndex system value semantic, then this value is used to select from the currently bound array of viewports. If this system value semantic is not present in the input signature, then the system defaults to the viewport located at index 0. Therefore, if only a single viewport is bound and no further manipulation of the viewport is needed, the developer can simply exclude SV_ViewportArrayIndex from the rasterizer input to select the primary viewport. Figure 3.58 demonstrates the use of multiple viewports and how they can be employed to select multiple regions for rasterization.

After the viewport transformation has been completed, the vertex position is split into its *X*- and 7-coordinates, which are used to identify the region of the render target that is covered by a primitive, and its Z-coordinate, which is a depth value that is used later in the depth buffering system to determine the visibility of individual fragments.

### Rasterization

The final operation in the rasterizer stage is to perform the actual rasterization of the primitives that reach this point in the pipeline. The primary purpose of the rasterizer is to convert the geometric data that it receives into discretely sampled data that approximates the geometric data within the render target. This can be thought of as a sampling process, where the geometry can be considered as continuous data, while the output fragments are a digital representation of the geometry at regularly spaced intervals. A number of sample primitives are shown in Figure 3.59.

The rasterization process consists of two different tasks, fragment generation and attribute interpolation. We will discuss each of these items individually, and then

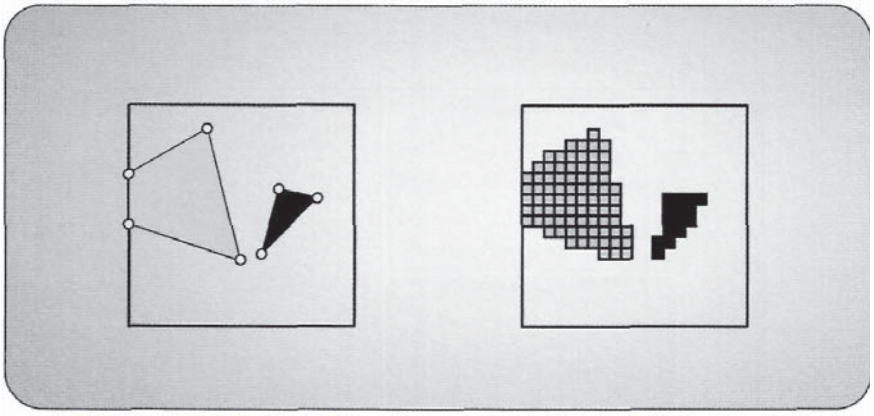


Figure 3.59. Various rasterized geometric primitives.

also look more closely at some of the fine details of using this process with multisample anti-aliasing (MSAA) enabled.

**Fragment generation.** The first step in rasterization is to determine which pixels of the render target are considered to be covered by the current primitive. This operation is performed by using a set of rasterization rules, which vary depending on the type of primitive that is being rasterized, as well as on the type of render target. If a render target supports MSAA, the rasterization process is different than if a standard render target is used. We will discuss the implications of MSAA later in this section, after the standard concepts have been introduced. With a grid of pixels representing the render target, the rasterizer stage must determine which of the pixels are "covered" by a primitive. Quotation marks were used here because there are many cases in which a pixel location is selected for rasterization but it is not completely covered by the primitive.

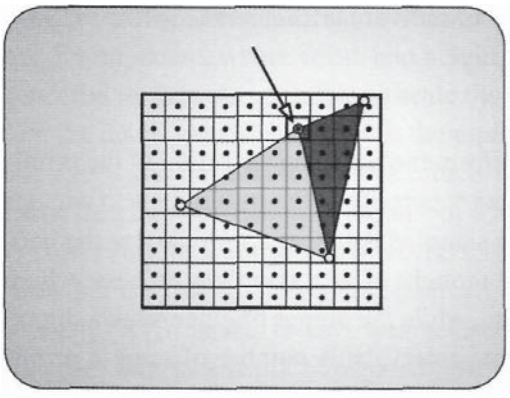


Figure 3.60. Pixels at the boundaries of multiple primitives.

To understand how this can happen, we first need to explore how fragment generation is performed. We will begin with the triangle rasterization process. Each pixel of the render target uses a point at the center of the pixel as the key for rasterization. Since it is at the center of the pixel, and each pixel is 1 unit large (in screen space), the center of the pixel is 0.5 units by 0.5 units from the edge of the pixel boundary. When a pixel center is completely covered by a primitive, it is clearly selected to produce a fragment. Since the pixel center can also land on one of the boundary edges of a primitive, some

additional rules must be used in testing to ensure that all pixels in two adjacent triangles are selected, and that no pixels that are selected more than once by neighboring primitives. Figure 3.60 demonstrates some of these situations.

The triangle rasterization rules specify that any points that fall onto a top edge or left edge of a primitive are selected for rasterization from that primitive. A top edge is any perfectly horizontal edge, while a left edge is any edge that faces to the left at any angle. If you consider the normal vectors of the range of edges that will allow selection of a pixel, the angles that will produce a fragment when an edge resides on a pixel center would appear as shown in Figure 3.61.

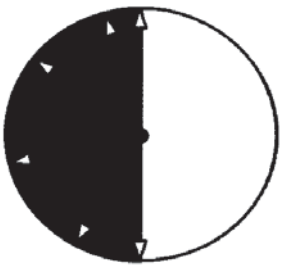


Figure 3.61. The angular range of primitive edge normal vectors to select a pixel for rasterization.

Line rasterization is somewhat different from triangle rasterization. It can follow one of two paths—either aliased or anti-aliased. Aliased line rasterization uses a fairly simple algorithm to determine which pixels are covered by a line, while anti-aliased line rasterization performs a more sophisticated algorithm to determine how much of a pixel is covered by a line, and then multiplies the output color for that pixel by this coverage factor. Selection of aliased or non-aliased line rasterization is determined by the AntialiasedLineEnable parameter of the rasterizer state. We will examine the details of aliased rasterization, while leaving the anti-aliased lines. The anti-aliased line algorithm implementation is hardware specific, and hence it does not make sense to try to interpret in great detail here.

Aliased line rasterization is performed as follows. Instead of using a point-based algorithm, the process starts by inscribing a diamond shape on each pixel. This is shown in Figure 3.62. If a line's slope falls within the range *-1 <= slope <= 1*, it is considered to be an *x-major* line. In this case, if the line itself intersects the bottom left or bottom right edge or the bottom corner of the diamond, then the pixel is selected for fragment generation. If

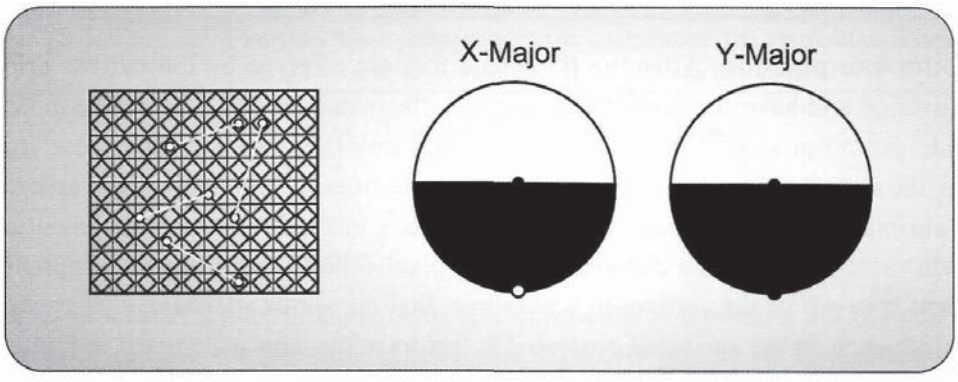



Figure 3.62. Some examples of line rasterization, with the two diamond selection possibilities depicted at the right.

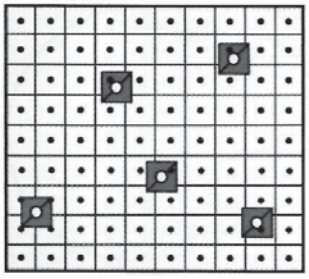


Figure 3.63. Several examples of point rasterization as a pair of triangles.

the line has a slope outside of the range defined above, then it is considered a *y-major* line. In this case, the same test is performed, with the exception that the right corner is also included in the test. These intersection images are shown in the right side of Figure 3.62.

The final primitive type that can be rasterized is the *point primitive*. Point primitive rasterization actually uses the same rules as triangle rasterization, by conceptually expanding itself into two triangles that form a 1 x 1 square around its point position. These triangles are then rasterized exactly as if they were triangle primitives. Several examples of this pattern are shown in Figure 3.63.

**Scissor test. In** all of these cases, the primitives must pass the *scissor test,* in addition to fulfilling the rasterization rules. As we have seen in the "Rasterizer State Configuration" section, the application can configure the scissor test by binding an array of scissor rectangles to the rasterizer stage. The rasterizer stage then selects the appropriate scissor rectangle, based on the same criteria used to select the viewport. If the SV_ViewportArrayIndex is declared as one of the input attributes for the rasterizer stage, then it is used to select from the array. Otherwise, the rasterizer defaults to the first entry. This selection mechanism ensures that the viewport index and scissor rectangle index are always the same, which allows the application to trivially reference pairs of these objects together.

The scissor test works by comparing the fragment's *X*- and *Y*-components against the scissor rectangle. If the fragment falls outside of the rectangle, it is culled and will not contribute to the render target. It is also important to note that the exact location of the scissor test in the sequence of operations of the rasterizer may vary, depending on the hardware implementation. This may have some performance implications, due to a variable location that fragments are culled from. However, the application can be sure that the pixels outside of the scissor rectangle will not be written to the render target.

**Attribute interpolation.** After the fragments that are affected by the current primitive are identified and have survived the scissor test, the rasterizer stage must determine what attribute data to produce for each fragment that it generates. To calculate these attribute values, the rasterizer interpolates each input attribute from its input primitive vertices. The input attribute value from each primitive contributes to the interpolated fragment output attribute value based on the distance from the pixel center. This means that the closer a fragment is to one of the vertices in a primitive, that the vertex will have a proportionally larger influence on the attributes generated at that location. This is depicted in Figure 3.64 with a simple example. In addition to the general attributes being interpolated, the depth at each fragment is also interpolated and will be used later in the pipeline in the depth test.

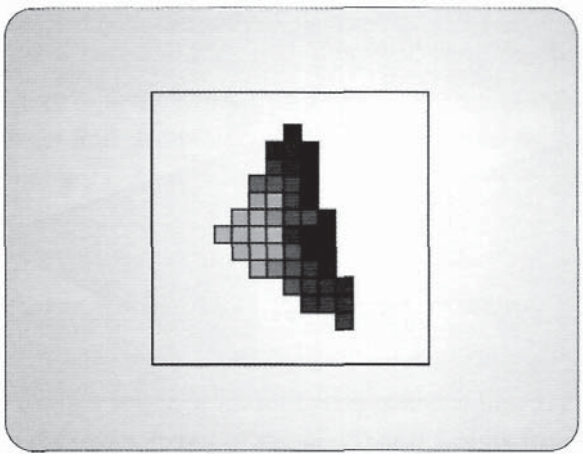


Figure 3.64. Several pixels of a rasterized triangle, which demonstrate how pixels are affected by their proximity to each of the vertices.

By default, the type of interpolation used to generate these per-fragment attributes is a perspective correct linear interpolation. However, the interpolation mode can be modified by adding an interpolation modifier before the type declaration of a pixel shader input attribute. The available interpolation modes are listed in Table 3.2, with a brief description of their behavior.

| Interpolation Mode | Description |
|---|---|
| linear | Provides linear, perspective-correct interpolation. The interpolation is based on the center of the pixel. |
| centroid | Provides linear, perspective-correct interpolation. The interpolation is based on the centroid of the covered area of the pixel. |
| nointerpolation | Provides no interpolation. Attributes are passed as constants. |
| noperspective | Provides linear interpolation without accounting for perspective effects. The interpolation is based on the center of the pixel. |
| sample | Provides linear, perspective-correct interpolation. The interpolation is based on the MSAA sample location, instead of the center of the pixel. |

Table 3.2. The available attribute interpolation modes.

## Multisampling Considerations

Direct3D 11's rasterization pipeline is an efficient means of rendering 3D geometry with performance suitable for real-time applications. However one of the fundamental drawbacks of rasterization is that the image produced can suffer from noticeable *aliasing* artifacts. Aliasing is a term from the field of signal processing that refers to the effect that

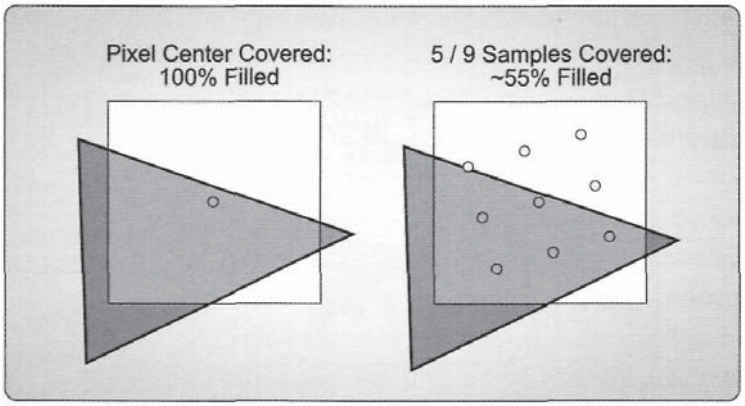



Figure 3.65. Rasterization sample points for non-MSAA and MSAA Rendering.

occurs when a continuous signal isn't sampled at a high enough sampling frequency to reproduce the original signal. If we were to describe rasterization in such terms, the "signal" would be the vector representation of our 3D geometry, while our "sampling frequency" is the *X* and *Y* resolution of the render target. In more general terms, we can say that aliasing occurs because our geometry must ultimately be rendered to a discrete grid of rectangles, using a binary coverage test. A grid of rectangles can never perfectly represent edges that are not completely horizontal or vertical, so a jagged "stair step" pattern occurs. This type of aliasing artifact is often referred to as *edge aliasing,* since it occurs at triangle edges. Another type of aliasing results from the fact that a pixel shader is only executed once for each pixel, thus discretely sampling the surface color resulting from the material and BRDF properties. This type of aliasing is referred to as *shader aliasing.* These artifacts can be very displeasing to the eye if the resolution of the image is small relative to the size of the display.

The classic signal processing approach to combat aliasing is to sample at a higher frequency, and then apply a low-pass filter. In rasterization, this roughly equates to rendering to a higher-resolution render target, averaging adjacent pixels, and using the result of the average as the pixel values for a normal-sized render target. In real time 3D graphics, this process is referred to as *super sampling anti-aliasing,* or *SSAA.* Increasing the resolution increases the sampling frequency used for both rasterization and pixel shading, thus effectively reducing the appearance of both types of aliasing. However, the additional performance cost of doubling or quadrupling the render target resolution often makes the technique prohibitively expensive. Because of this, Direct3D 11 includes a more simplified and optimized form of anti-aliasing known as *multisample anti-aliasing (MSAA).*

The basic premise of MSAA is that the resolution used for rasterization and the depth/stencil test is increased by an integral factor, but the pixel shader is still executed at the normal resolution of the render target. This allows it to effectively deal with edge aliasing, but not with shader aliasing. Since the depth and stencil tests are performed at a higher resolution, the depth-stencil buffer must store multiple subsamples per-pixel, causing the

memory footprint to increase by the MSAA factor. The same applies to the render target, since it must store individual color values for each sample before they can be resolved to a single value. Because of this, the MSAA sample count must be explicitly specified when creating the render target and depth stencil resources, so that the runtime can allocate a sufficient amount of memory. Also, the MSAA sample count must be exactly the same for render targets and depth-stencil buffers that are simultaneously used for rendering. Additional details for creating the resources needed for MSAA can be found in Chapter 2.

**Rasterizer interaction with MSAA.**When MSAA is enabled for rendering, the first modification to the pipeline is in the rasterization stage. Normally, during rasterization a triangle is tested for coverage using a single point for each pixel as outlined above. In Direct3D 11, this point is located in the exact center of the pixel, or 0.5 units from the upper-left corner in both the *X* and *Y* directions. This point is also used for interpolating all vertex attributes output from the previous stage (either the vertex shader, domain shader, or geometry shader, depending on which stages are in use), including the depth used for performing the depth test. When MSAA is enabled, multiple sample points within a pixel are used to test for coverage. The location of these sample points is implementation-specific, although typically these samples are arranged in a rotated grid pattern. An example of the sampling points for MSAA and non-MSAA rasterization are shown in Figure 3.65.

When a depth stencil target and MSAA render target have been bound to the pipeline, the rasterizer behavior is determined by the multisampleEnable member of the rasterizer state. This parameter essentially toggles whether the rasterizer performs multiple coverage tests for MSAA render targets. If it is disabled and an MSAA target is bound, then only a single coverage test is performed at the pixel center. If the coverage test passes, the pixel shader result is written to all subsamples. If it does not pass, then none of them are written to. This selectable MSAA operation allows a portion of a scene to be rendered using the multi-sampling functionality, while other portions of the scene that don't require anti-aliasing (such as 2D sprites for a user interface) can be rendered in the normal fashion. This produces a consistent appearance for these scene elements that don't use anti-aliasing, and also improves performance by reducing the number of coverage tests that are needed.

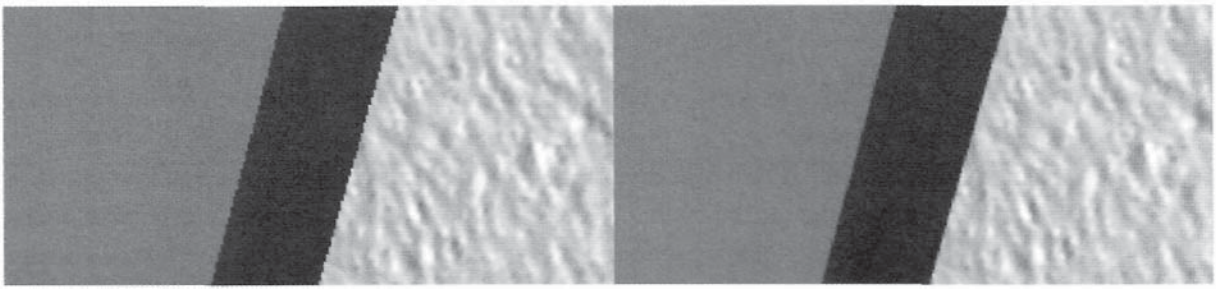


Figure 3.66. The difference between a rendering with and without MSAA enabled.

The difference between primitives rasterized with and without MSAA active is shown in Figure 3.66. You can clearly see the difference in how the edges of the primitive are significantly smoother in the MSAA version (shown on the left) than the non-MSAA version (shown on the right).

## 3.10.4 Rasterizer Stage Pipeline Output

### Fragment Generation

Throughout the rasterizer stage discussion, we have seen each of the processes that are used to convert a primitive into a set of fragments. Several different tests, such as culling, are intended to reduce the number of primitives that need to be processed. In addition, there are also several tests, such as clipping, viewport transform, and the scissor test, that potentially reduce the number of fragments generated by a primitive. It appears that there have been significant amounts of effort put into reducing the number of fragments generated by the rasterizer stage. Why would so many tests be aimed at reducing the output from this stage?

The reason is that the rasterizer stage has a 1-to-many relationship between its input and its output. For each primitive submitted to the rasterizer as input, many fragments are potentially generated. Due to this general data amplification, any processing performed after this point in the pipeline will be executed many more times than processing performed earlier. Therefore, it is clearly better to eliminate unused or unnecessary primitives before they are split into fragments, and it is also better to eliminate fragments in the rasterizer stage before they need to be processed later on. This idea can also be extended to general algorithm design, as well. When you have a choice between performing a calculation before or after rasterization, it is typically more efficient to perform the calculation before generating many fragments. In many cases, even if the calculations done before the rasterizer are only approximations of those that would be done afterward, this can produce a large efficiency gain at a minimal loss of image quality.

### Fragment Data

With this in mind, we can consider what type of data stream is produced by the rasterizer stage. We have discussed in detail how each fragment is generated when its location within a render target is covered by a primitive. When it is determined that a location is covered by a primitive, each of the attributes that are passed into the rasterizer are interpolated, using the center of the pixel as the input to the interpolation function.[32] The depth value is also

[32] For MSAA rasterization, the sample locations may be used for interpolation as well.

interpolated and passed along with each fragment for use in the depth test of the output merger stage.

The position of the fragment may or may not be added into the generated fragments. This depends on whether the pixel shader input signature declares one of its input attributes to have the SV_Position system value semantic. Prior to the rasterizer stage, the SV_Position value was used to indicate a version of the vertex position. When passed into the rasterizer, it must contain the clip space vertex location. However, the *fragment* location is provided in the output of the rasterizer. This can be used in the pixel shader to look up values in other texture resources, or for performing processing based on the screen location, such as fading the render target color to black at the outer edges of the screen.

## 3.11 Pixel Shader

After a primitive is converted into fragments by the rasterizer, the fragments are passed to the pixel shader stage. The pixel shader stage is the final programmable shader stage in the rendering pipeline. It processes each fragment individually by invoking its pixel shader program. Each pixel shader invocation operates in isolation—no direct communication is possible between individual fragments being processed.[33,34] After the pixel shader has completed, the result is a processed output fragment, which is passed to the output merger stage. The location of the pixel shader within the pipeline is highlighted in Figure 3.67.

The pixel shader stage is responsible for the primary appearance of a rendered primitive. Prior to rasterization, processing was primarily focused on manipulating the number of primitives, their size, shape, and attributes. These are all geometric-style operations. During rasterization, these primitives are used to select which pixels are affected by them, and fragments are generated accordingly. The pixel shader can only receive the fragments that are passed to it—it is not possible to change the location of a fragment from within the

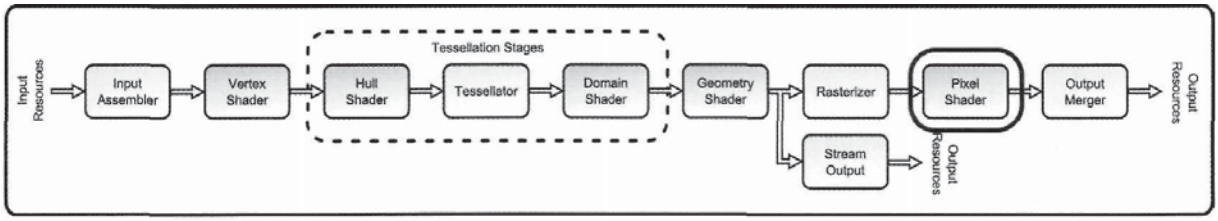



Figure 3.67. The pixel shader stage.

[33] The derivative instructions used data shared between multiple fragments.

[34] Unordered access views change this strict isolation, but there is still a fairly strong communication separation between pixel shader invocations.

pixel shader stage. Instead, the pixel shader determines how each of the selected fragments should appear, based on the fragment's attributes, and the information provided to the stage in various resources. This gives the pixel shader a relatively important role to play in the generation of images in real-time rendering.

The pixel shader stage also has some new abilities that were not available prior to Direct3D 11. It can use the new resource view type, the *unordered access view (UAV),* to perform read and write operations to the complete resource attached to the view.[35] This is a big departure from the traditional capabilities of the pixel shader stage, which could only write color and depth data into the pixel location that was passed to it. This provides the potential for a large number of algorithms to be implemented directly in the rendering pipeline, such as histogram generation, while the image is being rendered.

## 3.11.1 Pixel Shader Pipeline Input

The pixel shader stage receives its input fragment from the rasterizer stage. This means that the input attributes that the pixel shader program will use are produced by the rasterizer as well. We have already seen in the rasterizer section that it produces interpolated attribute data, based on the sampling location of a fragment within the primitive being rasterized, which is typically the center of the pixel (although sometimes from other sample locations). We also saw that various interpolation modifier keywords can be specified in the pixel shader program that will instruct the rasterizer to use a particular interpolation mode for each input attribute. These interpolation modes are useful in different scenarios, and must be chosen appropriately to ensure that the input attributes are calculated properly. We will explore these interpolation modes in more detail here.

### Attribute Interpolation

The process of interpolating attributes requires three different pieces of information. The first is the data that will be interpolated—the attributes of the vertices of the primitive being rasterized. This includes both their positions and their attribute values. The second piece of information needed for interpolation is the location of the point that requires the interpolated attribute data. This more or less determines how much of each vertex attribute the interpolated value will receive. The final piece of information required is the interpolation technique that should be used. This final item performs the actual interpolation on the first two items described above. The interpolation mode is determined by interpolation modifier declared with the pixel shader input signature.

[33] A resource view can be used to expose only a subportion of a resource instead of the complete resource, as described in Chapter 2.

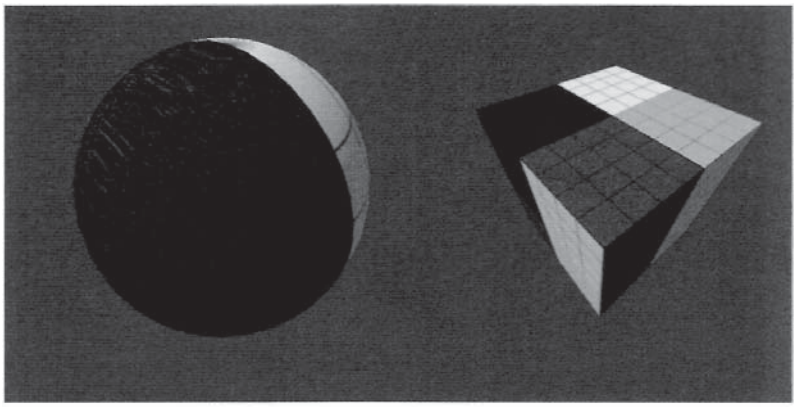


Figure 3.68. A sample of using the linear interpolation mode.

**The linear mode.** When the linear interpolation mode is used, the attributes that are generated for each fragment will be linearly interpolated, taking perspective effects into account. The need for perspective-correct interpolation arises from the fact that the perspective projection is not a linear operation, and this interpolation is performed after the projection. This means that standard linear interpolation can't be used between vertices in screen space, but instead, the depth of each vertex must also be taken into account.

The perspective effect can be accounted for in the interpolation process by dividing each vertex attribute by its depth from the viewer before interpolation. The depth value is found for each vertex as its (*W*-value, after projection, but before the conversion to normalized device coordinates. The reciprocal of *W* is also calculated, and then interpolated, along with these modified attributes. After all of these values are interpolated for each rasterized fragment, the interpolated reciprocal of *W* is used to extract the original desired attribute, which will produce a perspective-corrected interpolation value. This is the most common interpolation mode, with texture coordinates on a three-dimensional model providing a perfect example of when they are needed. Perspective-correct interpolation is needed not only for texture coordinates, but is also needed if a position or direction vector (or any other linear attribute) from a linear space (such as world space or view space) comes into the post-projection stages. An example of the linear interpolation mode is shown in Figure 3.68.

**The noperspective mode.** When the noperspective interpolation mode is used, the interpolated attributes are strictly interpolated according to their two-dimensional position on the render target. In essence, you can consider the geometry as being projected onto the render target before the interpolation is performed. This interpolation mode implements standard linear interpolation between vertices based on their positions. The interpolation of attributes without perspective correction can be performed whenever the vertices being interpolated all are at the same depth, such as when constructing an onscreen user interface rendering. Figure 3.69 demonstrates the use of this interpolation mode.

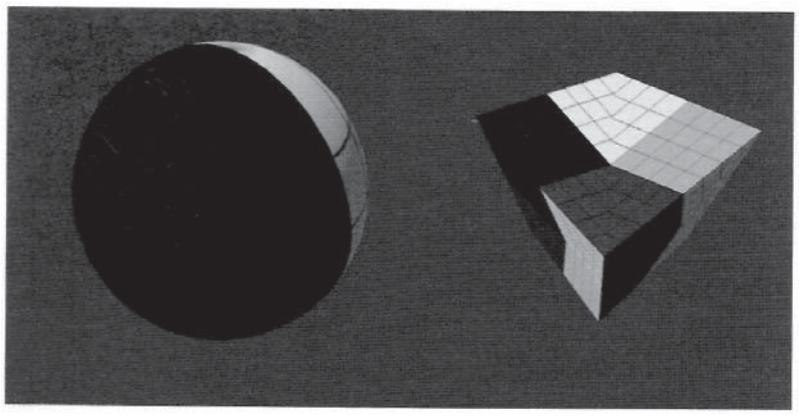


Figure 3.69. A sample of using the noperspective interpolation mode.

The **nointerpolation mode.** As its name implies, the nointerpolation mode does not perform any interpolation of the vertex attributes. This means that the attribute value of the first vertex of each primitive will be passed to all fragments for a given primitive, which produces a constant value over the entire surface of the primitive. This can be used to give a model a faceted appearance when this interpolation mode is used to modify lighting related attributes. In effect, not using interpolation will generally make the faces of the rendered geometry more noticeable, since the whole purpose of using interpolation is to hide the fact that the vertex-based geometry is approximating a smooth surface. This may be minimized when using the tessellation system, but could also be a good way to visualize how finely tessellated the final geometric results are. Figure 3.70 demonstrates the use of this interpolation mode.

**The centroid mode.** The centroid interpolation mode is intended to offer a more appropriate interpolation mode for special cases in MSAA rendering modes. When a pixel is

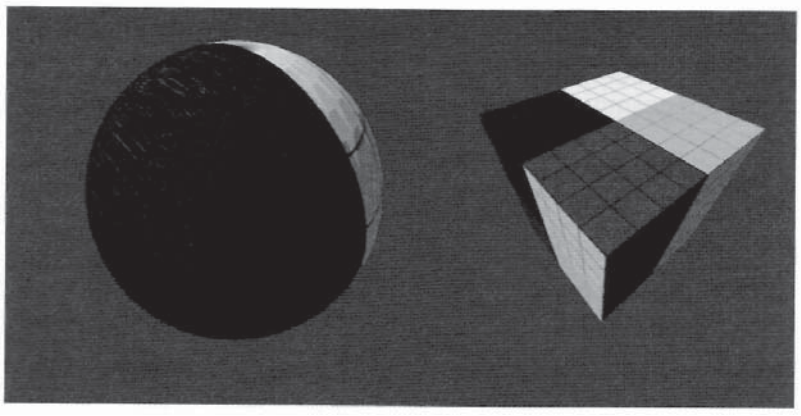


Figure 3.70. A sample of using the nointerpolation interpolation mode.

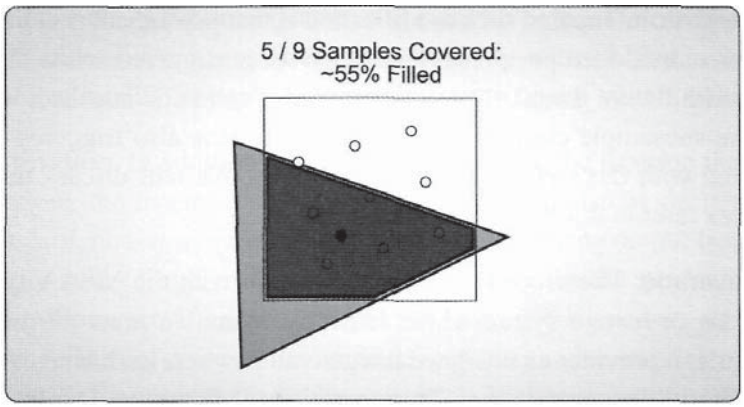



Figure 3.71. Interpolation point for centroid sampling.

partially covered by a primitive and MSAA is enabled, the rasterizer will produce a fragment for that pixel. However, if the primitive only covers a few samples within the pixel, but not the center of the pixel, then the normal `linear` interpolation mode would still use the center of the pixel as the input location to the interpolation function. Because of this, it is possible for the vertex attributes to be extrapolated beyond the true edge of the triangle, in cases in which only a portion of the sample positions are covered. For situations where extrapolation is undesirable, centroid sampling can be used on a per-attribute basis by adding the `centroid` modifier to the input variable declaration. When it is used, centroid sampling causes the affected attribute to be interpolated to a point that is guaranteed to be covered by the triangle. The location used is implementation-specific, but is generally located at the average of the covered sample points. An example of the centroid sampling mode is shown in Figure 3.71.

**The sample mode**. The sample interpolation mode provides interpolation at each of the sample points used in MSAA rendering mode. This is useful when the pixel shader is executed for each MSAA sample to provide attributes at each individual sample location. The topic of MSAA and how it relates to the pixel shader is discussed further in the following section.

### Multisample Anti-Aliasing

Several additional system value semantics can be declared as inputs to the pixel shader to offer some interesting possibilities for controlling its behavior when MSAA is enabled. Direct3D 11 offers several ways in which the "standard" MSAA behavior can be altered. These changes are based on the pixel shader bound to the pipeline, and on how it declares its input and output attributes.

**Per-sample execution.** The first such modification is that pixel shaders can be run for each subsample, rather than once per pixel. This behavior is triggered when the pixel shader takes an input with the SV_SampleIndex system value semantic attached, which provides the index of the subsample currently being processed. It is also triggered if an input attribute is marked with the sample interpolation mode. We will discuss the usage of the SV_SampleIndex later in this section in more detail.

**Subsample coverage.** The second possible interaction with the MSAA system involves the use of the SV_Coverage system value semantic. When an input attribute is declared with this semantic, it provides an unsigned integer value, where each bit corresponds to one of the subsamples of the current pixel. This essentially indicates to the pixel shader which subsamples were detected as being covered by the rasterizer stage during rasterization. We will see later how this system value semantic can also be used as an output attribute as well to perform custom modifications to the selected subsamples that will be written to with the pixel shader's results. This lets custom coverage masks be implemented, which is commonly used for alpha-to-coverage algorithms.

### Fragment Location

Two more system value semantics are accessible as inputs to the pixel shader to identify the location of a fragment, which is where it will ultimately be written to. These attributes can be used to aid the pixel shader program in processing a fragment.

**Source position.** When a pixel shader declares an input attribute with the SV_Position system value semantic, the pixel shader receives the four coordinate fragment positions produced by the rasterizer stage. In general, the most important parts of this position are the *X*- and *Y*- coordinates, which indicate the location of the fragment within the current render target. In most cases, these coordinates will indicate the pixel center location with the 0.5 pixel offset. This can be used to identify the location in other textures that need to be sampled when multiple textures are used to generate a pixel shader result. A common example of this is the use of multiple textures as a G-buffer for deferred rendering (see Chapter 11 for more details on deferred rendering).

In addition to the pixel center location, it is also possible to use this system value semantic to obtain the centroid position of the current fragment. As described above, the centroid location is guaranteed to be within the primitive boundary, and is normally used to ensure that the coordinates used for interpolation do not extend beyond the primitive's edge. The centroid value used in interpolation can be received if this semantic value is declared with the `centroid` interpolation modifier.

The depth value generated by the rasterizer stage is also available from within the SV_Position system value semantic. This is the normalized device coordinate depth, so its value will be in the range of [0,1]. This can be used to perform calculations within the

pixel shader that use this depth information, and can also be used to modify the rasterizer-generated depth value with a customized depth value. We will see later in this section how the pixel shader can modify its depth value using the SV_Depth system value semantic.

**Fragment destination.** In addition to being able to receive the location that the fragment was generated from, the fragment's ultimate destination can also be declared as an input. If there is a standard, non-array render target bound for receiving output from the pipeline, the output position of the fragment is more or less described by the SV_Position semantic discussed above. However, if an array-based render target is used to generate multiple simultaneous renderings of a scene, the render target array index must also be used to determine where the fragment will finally be written to. This is commonly used to generate various forms of environment maps, as demonstrated in Chapter 13.

The pipeline determines which render target slice to apply a primitive to by interpreting the SV_RenderTargetArrayIndex system value semantic. This parameter can also be used as an input attribute to the pixel shader. This lets the pixel shader program perform selective processing of a fragment that depends on the render target slice that will ultimately receive the fragment.

### Geometric Orientation

The final specialized input attribute that we can declare for input to the pixel shader is the SV_IsFrontFace system value semantic. This parameter provides an indication of the orientation of the primitive that generated the fragment. This is a Boolean attribute that is *true* when the fragment is generated by a front-facing primitive *and false* when it is generated by a back-facing primitive. You may recall that the rasterizer state can be used to determine which vertex winding defines the front-facing direction for a primitive, as well as which of these orientations are to be culled. If both orientations are allowed to be rasterized, this system value will specify which orientation was detected by the rasterizer stage.

Since point and line primitives have no notion of orientation, fragments generated from either of these primitive types will always produce a value of *true* for this input attribute. However, if a triangle primitive is rasterized in wireframe mode (which essentially generates lines from the edges of the triangles) the triangle orientation information is still used.

## 3.11.2 Pixel Shader State Configuration

With a clear understanding of what data the pixel shader program can receive, we will now consider what types of state configurations are available to the application. The pixel shader stage is a programmable stage, meaning that it has access to the standard common shader core functionality that we have seen throughout the various pipeline stages. Once again, we

won't repeat any code listings here, since they are essentially the same as we have seen in the vertex shader section. These common methods are listed below easy reference.

ID3D11DeviceContext::PSSetShader()

ID3D11DeviceContext::PSSetConstantBuffers()

ID3D11DeviceContext::PSSetShaderResources()

ID3D11DeviceContext::PSSetSamplers()

In addition to the standard resources of the common shader core, the pixel shader also has access to the new unordered access views, which provides random access to read and write to resources directly from the pixel shader. Finally, a function attribute called `earlydepthstencil` can be used to force the depth stencil test to be performed prior to the pixel shader. We will investigate these unique configurations in more detail in the following sections.

## Unordered Access Views

The pixel shader stage can use utilize unordered access views. As opposed to the shader resource views that provide *read-only* random access to resources, the unordered access views allow *read* and *write* random access. This lets the pixel shader program perform scatter writing operations, meaning that it can programmatically decide where to store the data it writes to a resource. Before having access to UAVs, the pixel shader could only write to the output render target and depth stencil render target in the location determined by the rasterizer stage when it generated the fragment.

The method for binding an unordered access view to the pipeline for use in the pixel shader is performed through the device context, just as it is for all other pipeline configurations. In this case though, the UAV is actually bound to the pipeline in the output merger stage instead of the pixel shader stage. This is done so that the render targets, depth stencil target, and any UAVs that will receive output from the pipeline are all bound for output from the pixel shader in the same method call. This also groups all possible pipeline output resources into a single pipeline stage, which somewhat simplifies the pipeline concept. In the output merger section we will see how resources are bound for use in the pixel shader with a UAV.

## Early Depth Stencil Test

The other unique configuration for the pixel shader stage is performed from within the pixel shader program's source code. The `earlydepthstencil` function attribute can be declared before a pixel shader program to indicate that the depth and stencil tests should be

performed before the pixel shader is executed (the depth and stencil tests will be covered in more detail in the output merger stage section). These tests are typically implemented to be executed as early in the pipeline as possible by the hardware implementation, but this attribute forces the tests to be performed early.

This is intended to improve the efficiency of a particular algorithm by eliminating fragments that would fail the depth test or the stencil test, and would thus be discarded. By doing so, any pixel shader invocations from fragments that would have failed the depth or stencil tests are prevented. This can be especially helpful when a scene has already been rasterized into the depth buffer and a second rendering pass is needed.

Another, more subtle situation that requires this attribute is when the rendering pipeline is only outputting data to a UAV through the pixel shader. In this case, the data is prevented from being written to the output merger, and hence the pipeline will never execute the depth or stencil tests. However, if these tests are still needed, it must be explicitly forced on with this function attribute.

## 3.11.3 Pixel Shader Stage Processing

We have now seen the types of information that the pixel shader stage receives from the rasterizer, and what additional resources it can be provided with to augment the available data. In this section, we will first consider some of the mechanics of how the pixel shader stage performs its duties. Next, we will further explore the pixel shader by considering how it can represent an object's material properties in a traditional rendering scenario. After this, we will look at some of the possible uses for the new UAV and consider some of the general operations that this new resource view type enables. Finally, we will complete this section with a discussion of how MSAA interacts with the pixel shader stage, and how to control the various MSAA functionalities to improve image quality while still retaining acceptable performance levels.

### Pixe! Shader Mechanics

We have seen how data is brought into the pixel shader stage. We can now consider some of the operations that the pixel shader will perform, and exactly what it must produce in a given pipeline configuration.

**Pixel shader execution.** Our conceptual operation model of the pixel shader stage is that when it is executing, each invocation processes a single fragment and then calculates an output color to pass to the output merger stage. Since the pixel shader operates after rasterization occurs, any calculations performed in the pixel shader are performed proportionally to the number of fragments that an object covers. Regardless of how complex the input geometry is, only the fragments within the viewport that the geometry covers are processed

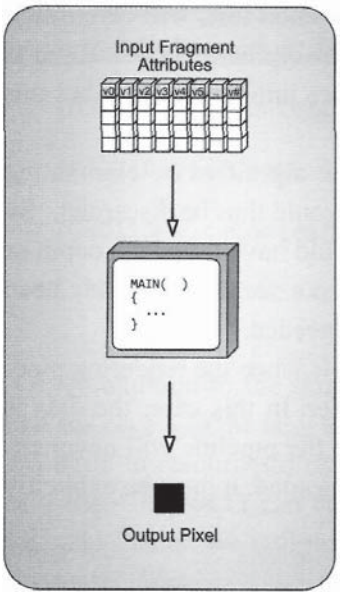



Figure 3.72. A pixel shader being executed to produce a single fragment output.

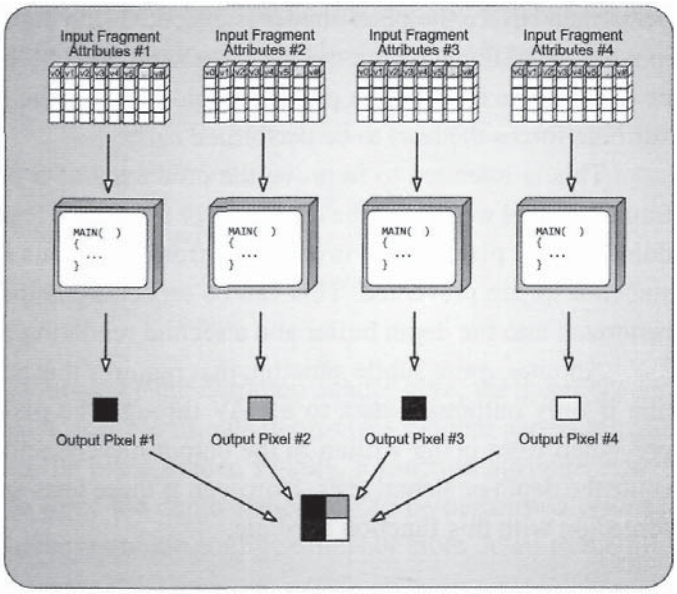



Figure 3.73. Many pixel shader invocations being executed simultaneously.

with the pixel shader. The concept of a single fragment being processed for each pixel shader invocation is depicted in Figure 3.72.

By following this model and restricting communication between threads, many different invocations of the pixel shader can be calculated simultaneously in parallel processing elements. Since each invocation receives its own data from its input fragment, and writes its own output to the location specified by the fragment, it does not rely on neighboring invocations. A schematic concept of this is shown in Figure 3.73.

At a lower hardware level, this is still mostly true, with the exception that pixel shader invocations are always performed in at least 2x2 groups of fragments. This is done to ensure that screen space derivative instructions can be calculated across the invocations in both the *X*- and *Y*-directions by performing a discrete difference. The calculation of these derivative instructions relies on the fact that multiple invocations are running at the same time, and if a variable is passed into one of the derivative instructions, then the GPU will use the difference of the neighboring invocations' variables to find how that variable's value is changing over screen space. In practice, this behavior is transparent to the application, and the boundary between pixel shader invocations remains.

**Multiple render targets.** We have already mentioned that the pixel shader calculates and outputs a color to pass to the output merger stage. However, this is not restricted to a single color. We will see in the output merger stage section that it is possible to bind up to eight

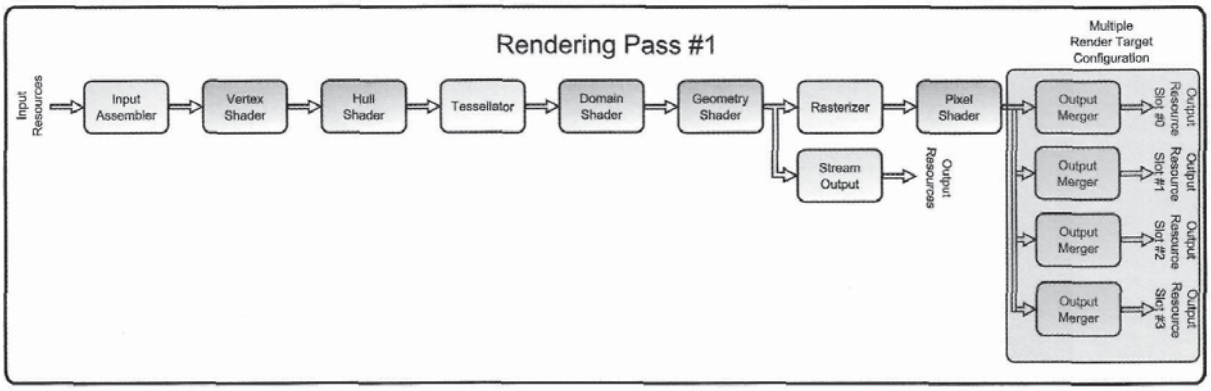



Figure 3.74. The difference between using MRTs and standard single render targets for filling several render targets with scene-based data.

different render targets for receiving output from the pipeline, as long as they meet certain size and format restrictions. A single pixel shader invocation calculates and outputs a color to write to each of these render targets. This ability to use more than one render target simultaneously is referred to as *multiple render targets (MRTs)* and provides the potential for improving the efficiency of a rendering algorithm.

If an algorithm requires multiple render targets to be filled with data produced from the scene's geometry, then it is possible to fill each render target one at a time using a traditional single render target configuration. This means that all of the calculations performed in the vertex shader, the tessellation stages, the geometry shader, and the rasterizer are repeated for each pass. The only calculations that are different between passes are those performed by the pixel shader. The use of MRTs allows all of the passes to be combined into a single one, after which the pixel shader writes to all outputs individually. This saves the cost of processing the geometric data multiple times, but still enables the same number of render targets to be filled with the desired information. Figure 3.74 visualizes the difference between using MRTs in this scenario and making multiple rendering passes instead. A very good example of the use of MRTs is provided in Chapter 11, "Deferred Rendering."

**Modifying depth values.** In addition to being responsible for writing the color values for a given fragment, the pixel shader also can output a new depth value to the SV_Depth system value semantic. If the pixel shader does not write the depth value, the depth generated in the rasterizer stage is passed to the output merger stage. However, in some cases it is more appropriate for the pixel shader to specify a depth value instead. A scenario where this would be useful is when using *billboards*. A billboard is commonly used to simulate more complex geometry with two triangles arranged to form a quad, which is then aligned to be perpendicular to the current view direction. A texture is applied to the quad, and the results

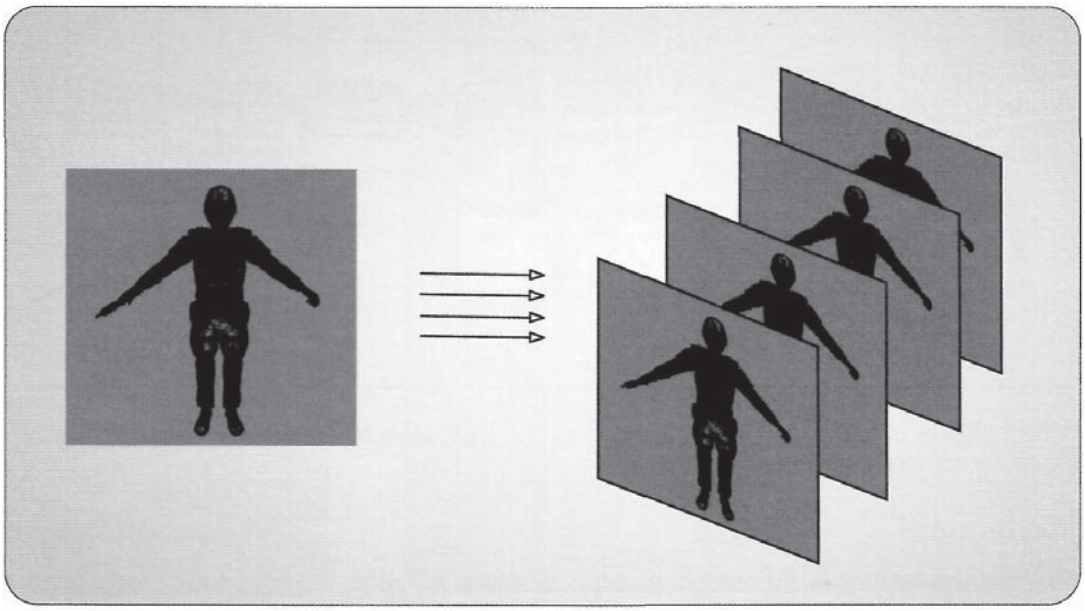


Figure 3.75. Using a billboard to simulate more complex geometry. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

of rendering just the two triangles still provide a significant amount of detail.[36] The concept of billboards is illustrated in Figure 3.75.

A billboard provides an increased amount of detail with very simple geometry, which is an efficient way to introduce complexity to a scene. However, since a billboard is essentially flat, when it is used in situations where it intersects other scene geometry, the illusion of the complex geometry is spoiled, because the billboard has a uniform depth across its surface. In this scenario, if the depth variations of the simulated geometry were included in the billboard texture (in the alpha channel for example), the pixel shader could write a modified depth value to the fragment, thereby reintroducing depth complexity to the billboard geometry. Then, when scene geometry intersects the billboards, they will actually appear more convincing, with partial occlusion rather than complete occlusion. An example of this type of depth modification is shown in Figure 3.76.

**Conservative depth output.** However, writing depth values from the pixel shader does have some drawbacks. Most modern GPUs implement an efficiency improvement technique referred to as *Hierarchical-Z Culling,* or *Hi-Z.* The concept behind this technique is for the GPU hardware to perform some simplified forms of occlusion tests to see if the current batch of geometry will be able to be seen in the final render target, or if it would appear behind another object in the rendering. If the geometry would be occluded, then it is simply

[36] Billboards are used in the particle system sample discussed in Chapter 12, "Simulations."

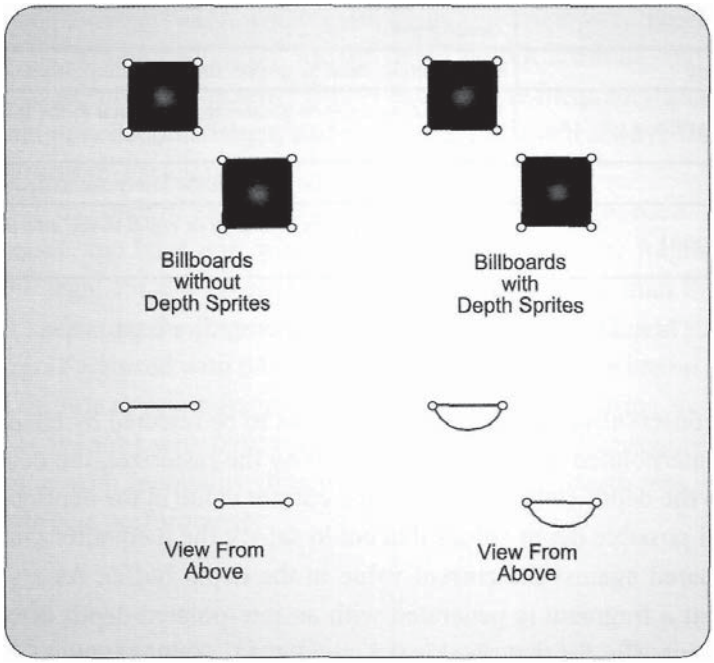



Figure 3.76. An overhead view of how depth can be added to a billboard to allow proper depth testing.

discarded before any further processing. This reduces the overall computational cost of the pipeline execution when there are many objects that overlap within a scene. The details about how this is implemented vary by GPU manufacturer, so they will not be covered in detail here. However, some of the assumptions made about the size of the geometry, which are used during this hierarchical testing, occur prior to the execution of the pixel shader. Therefore, when the depth value is modified in the pixel shader, Hi-Z may be unusable and its performance benefits can be lost.

To allow Hi-Z to remain active for a certain subset of algorithms that require depth output, Direct3D 11 introduces a new feature known as *conservative depth output*. This technique works by requiring pixel shaders to specify an inequality function along with the new depth value. The inequality effectively specifies an upper or lower bound on the depth output, which allows Hi-Z to continue to identify fragments that can be trivially rejected and thus don't require the pixel shader to be executed.

To have a pixel shader make use of conservative depth output, the shader program must assign one of four new system value semantics to the value used for outputting depth. These semantics each specify the inequality as part of the semantic name, and the depth value must satisfy that inequality relative to the interpolated depth value calculated by the rasterizer stage. If the pixel shader outputs a depth value that fails to satisfy the inequality, the runtime will automatically clamp the value to the appropriate minimum or maximum value. The four semantics are listed in Table 3.3.

| Semantic | Description |
|---|---|
| SV_DepthGreater | Depth output must he greater than the interpolated depth value |
| SV_DepthGreaterEqual | Depth output must be greater than or equal to the interpolated depth value |
| SV_DepthLess | Depth output must be less than the interpolated depth value |
| SV_DepthLessEqual | Depth output must be less than or equal to the interpolated depth value |

Table 3.3. System value semantics for conservative depth output.

Whether conservative depth allows a fragment to be rejected by Hi-Z depends on the inequality, the interpolated depth value produced by the rasterizer, the depth testing function specified in the depth-stencil state, and the current value in the depth buffer. For rejection to occur, all possible depth values that could satisfy the inequality must fail the depth test when compared against the current value in the depth buffer. As a simple example, let's suppose that a fragment is generated with an interpolated depth of 0.75, the current value in the depth buffer for that pixel is 0.5, and D3D11_COMPARISON_LESS is specified as the current depth comparison function. If a pixel shader without depth output is used, the Hi-Z hardware will determine that 0.75 is greater than the current depth value, and that

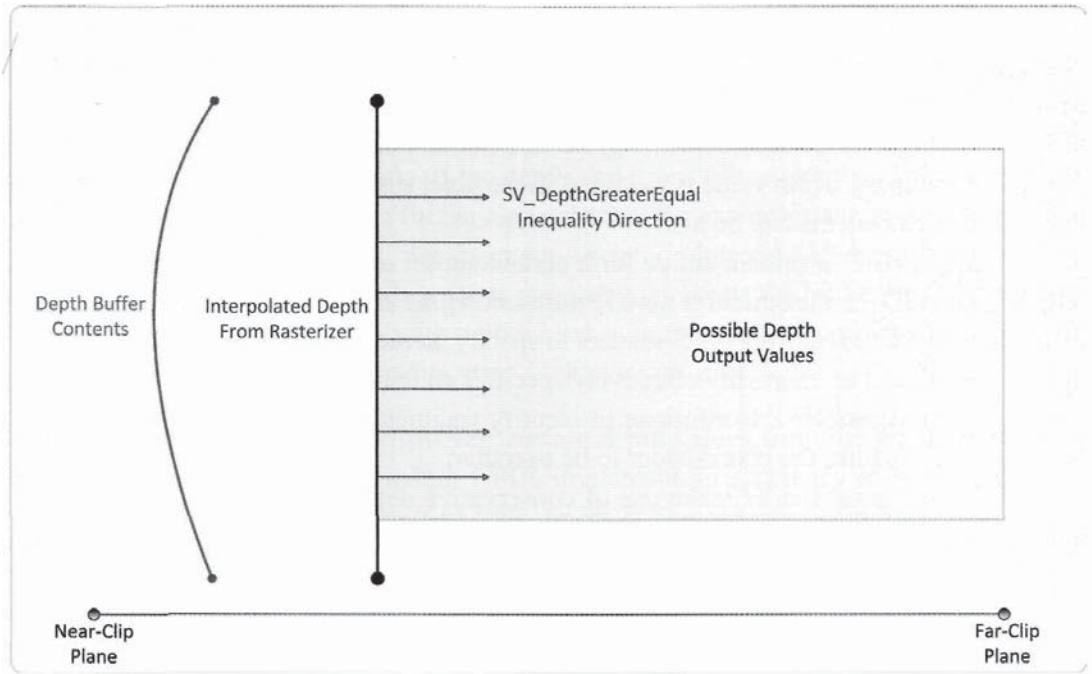



Figure 3.77. Conservative depth output with "SVDepthGreaterEqual."

the fragment should be rejected. If a pixel shader with conservative depth output is used that specifies SV_DepthGreaterEqual for the depth output attribute, the Hi-Z hardware will still be able to reject the fragment. This is because the inequality guarantees that the depth output could only become larger, and consequently, it will always fail the depth test. Figure 3.77 illustrates this scenario.

Note that if SV_DepthLess or SV_Depthl_essEqual were used instead of SV_DepthGreaterEqual, the Hi-Z unit would not be able to reject the fragment. This is because it would be legal for the pixel shader to output values less than the current depth buffer value of 0.5, causing the fragment to pass the depth test. In fact, if 5V_DepthLess or SV_DepthLessEqual are used with D3D11_C0MPARIS0N_LESS, there are no cases where the Hi-Z unit would be able to reject fragments. The same applies to using SV_DepthGreater or SV_DepthGreaterEqual with D3D11_C0MPARIS0N_GREATER. Consequently, a conservative depth semantic with inequality direction *opposite* of the depth test direction should always be used for best performance.

### A Small Example

Now we move on to consider how the pixel shader stage is used to implement a rendering model. The primary responsibility of this stage is to produce the output color that will be merged into the render target bound at the end of the pipeline. Thus, the ultimate decisions the developer must make are what type of calculation to perform, and what data inputs to use for that calculation. The types of algorithms can range from a single color output for all pixels that are rendered, all the way to a complete global illumination system that requires several simulation steps before the final rendering can be performed.

To explore how various algorithms can be implemented, we will first examine one simple rendering example and see how the pixel shader is used to formulate the rendered color output. Along the way, we will explain how this example scenario represents the basic methodology that the pixel shader operates with. By taking a high-level view of these concepts, we will be able to implement a wide variety of algorithms, including some that aren't even used to generate color data such as shadow maps. The details of the implementation aren't as important as the details of how processing is performed within the stage.

As mentioned above, the pixel shader is responsible for producing the final color values that will be merged into the render targets at the end of the pipeline. In this sample scenario, the color determination will be a product of the object's material properties, as well as of some environmental properties that the material interacts with, such as the presence of lights. An object with the same material properties can appear quite different when viewed in different lighting conditions, just as models with differing material properties can appear different even if they are being rendered in the same lighting conditions. Thus, both sets of properties must be made available to the pixel shader program to properly determine what color to produce.

**Material properties.** The material properties of an object determine how it looks at a basic level. Typical material properties include things like the base color of an object, some coefficients to describe how reflective the surface is, and perhaps a texture map that represents the fine surface color variations. We will assume that the pixel shader receives the vertex normal vector and texture coordinates as interpolated input attributes. The pixel shader program will calculate the color of the object's material with the function shown in Listing 3.23.

```
float4 PSMAIN( in VS_OUTPUT input ) : SV_Target
{
    // Determine the color properties of the surface from a texture
    float4 SurfaceColor = ColorTexture.Sample( LinearSampler, input.tex );

    // Return the surface color
    return( SurfaceColor * input.color );
}
```

Listing 3.23. A sample pixel shader to calculate the color of an object's material.

In a typical scene, many different objects need to be rendered, and each one will require at least partially different properties from the others. For this example, we will assume that the pixel shader is common among each of the objects being rendered. With this in mind, we can step through a few object rendering sequences and consider what is different between each one with respect to the pixel shader. We will assume there are three objects in the scene, called objects A, B, and C. For each of these objects to be rendered, the pipeline must be configured with the object properties prior to execution of the pipeline with one of the draw calls.

When the pipeline is configured to render object A, the object's texture is bound to the pipeline with a shader resource view, and its color is specified by binding a constant buffer to the pixel shader stage containing the appropriate color. Each of the generated fragments is passed to the pixel shader, where the texture is sampled and multiplied by the color supplied in the constant buffer, as shown in Listing 3.23. After this pipeline execution completes, the application would configure the pipeline for rendering object B. This would require binding a different texture and color constant buffer, followed by pipeline execution. Object C would follow the same pattern, but in this case, it would use the same texture as object B, and only a new color constant buffer would be needed.

This short sequence demonstrates that an object's material properties are typically supplied as resources external to the shader program itself. Both the color and texture of a model are controlled by the application manipulating the pixel shader stage's states, rather than by swapping the pixel shader program in and out. In addition, the geometry that is passed through the pipeline doesn't really matter to the material properties. In all three

cases, the input geometry could have been exchanged between objects, and the resulting appearance of the material would have remained the same. This is because the pixel shader operates after the geometric data in the pipeline has been converted into a rasterized form. It processes data after rasterization, and so is not affected by changes in geometry. While keeping these properties in mind, we will next consider how to have our example objects interact with their environment by adding lighting to the example.

Lighting **properties.** By providing lighting information in a scene, we can significantly increase its quality. Lighting is a very fundamental part of how we see the physical world, and using it significantly improves generated scene renderings. Many different types of lighting representations are used in modern real-time rendering, but for the purposes of this sample, we will use a simple directional lighting model. The light will be described by a direction vector and a color, both of which will be provided in a second constant buffer. The amount of light that reaches a given surface is commonly approximated by taking the dot product of the surface normal vector and the vector representing the light's direction of travel. This produces a scalar value in the range of [0.0,1.0] (as long as both the normal vector and the light vector are normalized) and can be used to scale the amount of light applied to a surface. A function that performs this operation is provided in Listing 3.24.

```
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 * input.color * Illumination );
}
```

Listing 3.24. A function that performs a simple lighting equation.

Now we can return to our sample renderings of objects A, B, and C. The rendering sequence will be repeated, except that the lighting information must be supplied to the pixel shader in a second constant buffer. Since each of these objects resides in the same scene, they all utilize the same light description. This means that the same constant buffer can be reused for all of the pipeline executions. To carry out the example, the pipeline is configured for each object in the same manner as before, with the addition that the lighting

constant buffer is also bound. When the pixel shader executes, the object's material color is found in the same way as before, and the amount of light visible at each pixel location is calculated, based on the normal vector passed into the pixel shader as an input attribute and the light direction vector. The result of the lighting calculation is then used to modulate the color of the fragment. As each additional object is rendered, the process is repeated exactly with each of its material properties, and the resulting rendering now incorporates lighting in addition to material colors.

In this case, we see that the lighting information is the same throughout the scene and hence is applied to each of our three objects in the same way. Environmental data is the same for all objects that share the same environment. We also see that even though each of the three objects has a different material appearance, they all interact with the light in the same way, using the same calculation regardless of what color they are.

**Generalizing the example.** So what have we learned from this example, and how can we apply the results of these simple experiments to understand the general concept of using the pixel shader to implement rendering techniques? The pixel shader program is simply a function that takes a certain number of input arguments and produces a color. Some of the inputs are changed at every pipeline execution, such as material properties, and some of the inputs are changed once per rendered frame, such as the lighting properties. Still others won't change at all throughout an application's lifetime, such as the vertex normal vectors of a model.

The flexibility provided by the pixel shader becomes quite clear now. It does not really matter what the function is that calculates the color of a generated fragment. As long as it produces a color result that varies appropriately when the input is varied, the rendering model serves its purpose. Developing a different rendering model revolves around deciding what inputs should be used to calculate the output color, developing the function that carries out the mapping from input to output, and then producing the geometric content that fits into the rendering model. More complex rendering models may require more inputs, including the possibility of using dynamically generated inputs such as the result of additional rendering passes. However, the pixel shader itself always resolves to a way to convert the input data to an output color.

### Using Unordered Access Views

The previous example demonstrated how the pixel shader can be used to implement a rendering model. In this section, we consider what kind of additional possibilities are made available by the inclusion of the unordered access views to the pixel shader. The UAVs allow the various resource types to be read or written at any location, by any invocation of the pixel shader. Before the addition UAVs, the pixel shader stage was restricted to only reading from resources, with the exception of writing its output color(s) and depth to the fragment location.

This new ability to write to any location of a resource can be used to extract additional information from the rendering process. For example, the generation of a histogram can be performed simultaneously with the rendering of the scene. A histogram provides a number of *bins,* where each bin represents the number of pixels that have a value that resides within a particular range. A histogram provides an indication of the distribution of the values in an image. By examining the output color of a pixel shader, the histogram bin that the color falls into can be determined. Then, a resource can be accessed using a UAV, and the appropriate bin value can be incremented. This would not have been possible prior to the introduction of UAVs.

Even more complex rendering algorithms can be implemented with the use of UAVs. Since read and write access to resource is available, generic data structures can be implemented using UAVs. This ability has been used to build linked list implementations out of GPU resources, which can then be used as building blocks for other algorithms.

## Multisample Anti-Aliasing Considerations

Since the pixel shader operates very close to the final rendered image, several special pixel shader functionalities should be considered when using MSAA. With non-MSAA rendering, the coverage test performed for each pixel's sample point determines if the pixel shader stage should be executed for that pixel. With normal MSAA rendering, the pixel shader is still only executed once per pixel. Thus, the pixel shader will be executed as long as at least one of the sample points passes the coverage test. After the shader is executed, the value(s) output by the shader are written to the corresponding subsamples in the currently bound render target(s).

**Sampling MSAA textures in shatters.** In Direct3D 11, shader resource views can be created for MSAA textures, so that the textures can be bound as inputs to the various shader stages. This is done by setting D3D11_SRV_DIMENSI0N_TEXTURE2DMS as the ViewDimension member of the D3D11_SHADER_RESOURCE_VIEW_DESC structure passed to the ID3D11Device : :CreateShaderResourceView method. However, when the view is bound to the pipeline for a stage, that stage cannot sample the texture using the normal Texture2D object and its various methods. Instead, a Texture2DMS object must be declared. This object supports the load method, which allows you to specify both XY position and the index of the subsample that you want to retrieve. It also supports the GetDimension method for querying the XY dimensions and the number of subsamples in the texture, as well as the GetSamplePosition method for querying the pixel coverage sample point used for a specified subsample index. Using these methods in a pixel shader allows for custom resolve algorithms to be implemented, rather than relying on what the hardware provides.

**Alpha-to-coverage.** Alpha -to -coverage is a technique that modifies how a pixel shader's output is written to the rendertarget. When enabled by setting the AlphaToCoverageEnabled

member of the D3D11_BLEND_DESC structure to *true*, the output of the pixel shader is no longer written to the various subsamples solely based on the results of the rasterization stage coverage test. Instead, the mask generated by the coverage test is bitwise AND'ed with a mask generated from the alpha component of the value output from the pixel shader. This result is AND'ed with a screen-space dither mask, which causes the surface to be rendered with a fixed dithering pattern across the subsamples. The most common use of this feature is to implement a simplified version of transparency, where transparent surfaces are dithered based on their opacity, rather than blended. This provides a performance advantage, since the pixel output does not need to be blended with the render target contents, and also because it does not require transparent surfaces to be sorted based on their distance from the camera.

Using alpha-to-coverage doesn't explicitly require that MSAA be enabled. However, the quality is significantly improved when used in conjunction with MSAA, since it allows dithering to occur at the subsample level, rather than at the pixel level, providing a higher output image quality.

**Pixel shader behavior modifications.** Direct3D 11 offers several ways in which the "standard" MSAA behavior can be altered. These changes are based on the pixel shader bound to the pipeline, and how it declares its inputs and outputs.

**Per-sample execution.** The first such modification is that pixel shaders can be run for each subsample, rather than once per pixel. This behavior is triggered when the pixel shader takes an input with the SV_SampleIndex semantic attached, which provides the index of the subsample being currently shaded. This index corresponds to the index passed to Texture2DMS.Load, making it simple to retrieve the appropriate subsample from another MSAA render target. The per-sample behavior is also triggered if the "sample" modifier is attached to an input, which causes the attribute to be interpolated to the current sample point, rather than the pixel center.

Having the pixel shader run at a higher frequency means that shader aliasing can be reduced, in addition to edge aliasing, or that special behavior can be implemented for non-traditional rendering pipelines. However, it should be noted that having the pixel shader execute at sample frequency means that performance for those pixels is effectively reduced to that of supersampling, since the pipeline is essentially the same. Therefore, it is usually beneficial to only use per-sample shaders for a subset of the geometry rendered, or to use the stencil buffer to mask out pixels that don't require per-sample shading. For an example using per-sample pixel shaders, see Chapter 11, "Deferred Rendering."

**Depth output.** As described previously, the pixel shader allows a value to be manually specified for depth testing and depth writes, using the SV_Depth semantic. When MSAA is enabled and the shader is not executed per-sample, outputting to SV_Depth causes a single value to be used when testing depth for each sample point. This can cause MSAA to function incorrectly for pixels that are overlapped by multiple triangles.

Custom coverage masks. Under normal conditions, the value output from a pixel shader is based on the coverage mask generated by the coverage test performed in the rasterizer stage. However it is possible for pixel shaders to instead output their own coverage mask instead, using the SV_Coverage semantic. The value output is a *uint,* where each bit corresponds to a subsample in the render target. Setting a bit to 1 causes the output value to be written to a subsample, while setting it to 0 prevents it from being written. The obvious use of this semantic is for implementing a custom mask for alpha-to-coverage transparency, rather than relying on the fixed screen-space mask implemented in most hardware. Another possible use is for implementing MSAA is a *deferred Tenderer,* which is explored further in Chapter 11, "Deferred Rendering."

### 3.11.4 Pixel Shader Pipeline Output

Since the pixel shader stage sends its output to the end of the pipeline, the output merger stage, there are limitations on the output attributes that it can write to. The output merger can only process the color values and depth described above, and no additional information can be received. This is enforced by the shader compiler, which requires that any output attributes be limited to those system value semantics that represent color and depth, namely SV_Target[n] and SV_Depth.

The only exceptions to these attributes are the SV_DepthGreaterThan, SV_DepthLessThan, and SV_Coverage semantics. The first two provide the mechanism to continue to enable the hierarchical z-culling algorithm when the SV_Depth attribute is written to. Therefore, it is not useful to write either of these semantic attributes if the depth is not manually modified in the pixel shader. The SVCoverage semantic makes it possible to use customized subsample coverage masks, as detailed above. Since it specifies a pattern to be used in an MSAA render target, it also should only be used in situations that use MSAA render targets.

## 3.12 Output Merger

The final stop in the pipeline is the output merger stage. This is a fixed function stage that receives color and depth results from the pixel shader stage, and then merges those results into the render targets bound to it for output. However, this stage provides significantly more functionality than simply writing color and depth values to resources. The output merger also performs visibility determination with the depth test, which implements the traditional Z-buffer algorithm. In addition, it can also perform a stencil test to precisely control which areas of the render target are written to. We will examine how these two

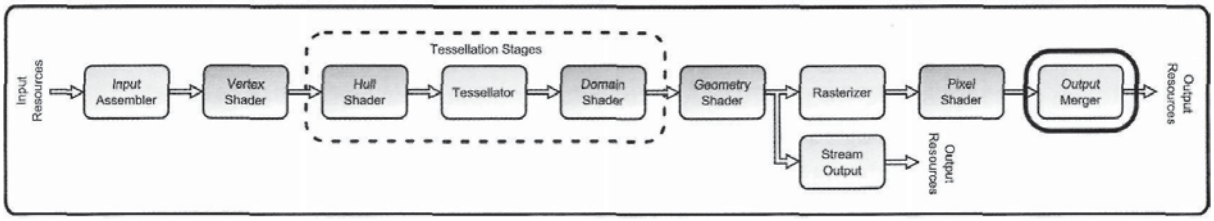



Figure 3.78. The output merger stage.

tests function in more detail later in this section. The location of the output merger stage is highlighted in Figure 3.78.

The output merger is also capable of modifying the color values that are passed on to it from the pixel shader through the blending function. This configurable functionality allows for a variety of different blending modes, which can be used to combine the current pixel shader results with the existing contents of the render target. This blending functionality, coupled with the ability to handle MSAA render targets, multiple render targets, and render target array resources makes the output merger play an important role in producing the final rendered image.

### 3.12.1 Output Merger Pipeline Input

Because it is the final stage in the pipeline, input to the output merger stage is the complete set of data produced by the pixel shader stage. The primary input for the output merger is the color values that were produced by the pixel shader program. The number of color values received depends on the output signature declared in the pixel shader, but in general, the number of output colors will match the number of render targets that have been bound to the output merger for an MRT configuration. There is one exception to this, which is when two colors are output from the pixel shader for use in dual-source blending. This concept is discussed further in the blending portion of the "Output Merger Stage Processing" section.

The second major input that the output merger stage receives is the fragment's depth value. This value is received either from the rasterizer stage, or the pixel shader stage if it is manually modified in the pixel shader program. These depth values are the input to the depth test, and will ultimately be used to update the depth buffer contents.

As described in the "Pixel Shader Pipeline Output" section, both of these values are output for each subsample when MSAA is used. In this case, an additional input system value semantic can be received from the pixel shader. The SV_Coverage semantic indicates which subsamples will be written to by the pixel shader output. This subsample selection process is performed automatically in the output merger stage and requires no additional effort by the application, but it is still important to understand how the output data will eventually end up in a render target.

## 3.12.2 Output Merger State Configuration

The output merger stage is controlled by a pair of state objects—the depth stencil state and the blend state. In addition to these, the output merger can also have resources bound to it, which ultimately represent the final output of the rendering pipeline. In this section, we will consider how each of these properties is configured, and what restrictions there are for handling each possible state and resource.

### Depth Stencil State

Both the depth test and the stencil test are configured with the same state object, the ID3DllDepthStencilState. As with all resources in Direct3D 11, this object is created through the device interface. The depth stencil state is created with the ID3DllDevice:: Create DepthStencilStateQ method, which takes a pointer to a description structure that contains the desired state options. Listing 3.25 lists the members of the depth stencil state description.

```
struct D3D11_DEPTH_STENCIL_DESC {
    BOOL                        DepthEnable;
    D3D11_DEPTH_WRITE_MASK      DepthWriteMask;
    D3D11_C0MPARIS0N_FUNC       DepthFunc;
    BOOL                        StencilEnable;
    UINT8                       StencilReadMask;
    UINT8                       StencilWriteMask;
    D3D11_DEPTH_STENCIL0P_DESC  FrontFace;
    D3D11_DEPTH_STENCIL0P_DESC  BackFace;
}
```

Listing 3.25. The D3D11_DEPTH_STENCIL_DESC structure and its members.

The first three parameters of the structure configure the depth test functionality, while the remaining parameters are used to configure the stencil test. Once the depth stencil state has been created, it is immutable and cannot be changed. If the application requires another state to be active, a separate state object must be created and bound to the pipeline in the place of the current state. This is accomplished through the ID3DllDeviceContext::OMSet DepthStencilStateQ method. As usual, there is a corresponding get method to retrieve the current state that is bound to the pipeline. The details of what these members do are discussed in more detail in the "Output Merger Stage Processing" section.

### Blend State

The second state object that the output merger supports is the ID3D11BlendState. This blend state object controls how color values are blended together before being written to

the output render targets. This stage object is created with the ID3DllDevice:: Create BlendState() method, which takes a pointer to a description structure that specifies what configuration the state object will represent. Listing 3.26 shows the contents of the blend state description.

```
Struct D3D11_BLEND_DESC {
    BOOL                           AlphaToCoverageEnable;
    BOOL                           IndependentBlendEnable;
    D3D11_RENDER_TARGET_BLEND_DESC Render-Target[8];
}
```

Listing 3.26. The D3D11_BLEND_DESC structure and its members.

The first parameter determines if the alpha value of the output color will be used to determine the subsample coverage instead of the rasterizer coverage values, or the SV_ Coverage system value semantic if the pixel shader writes to it. This can be used to implement a form of transparent rendering without needing to first sort the objects in the scene. The second and third members of this structure are used in conjunction with one another. The IndependentBlendEnable parameter indicates if there will be a separate blending mode specified for each render target when multiple render targets are used. If the parameter is *true,* each of the eight elements in the RenderTarget array provides the blending state for its respective render target. If independent blend is *false,* the element at index 0 will be used for all render targets. Once the blend state object has been created, it is bound to the pipeline through the device context with the ID3DllDeviceContext: :OMSetBlend State() method. And as always, there is a corresponding get method to retrieve the state currently bound to the pipeline. We will explore the details of the blending configurations in the "Output Merger Stage Processing" section.

### RenderTarget State

The render target state of the output merger represents the actual resources that receive the results of all the rendering calculations performed throughout the pipeline. A relatively large number of different configurations can be used to receive this output, and of course there are also restrictions on what combinations of render targets can be used together. We will begin by discussing the various types of render targets that can be bound to the output merger and then see how they are manipulated by the application.

The output merger has eight render target slots and one depth stencil slot. Each render target is bound to the pipeline through a render target view *(RTV),* and the depth stencil target is bound through a depth stencil view *(DSV).* In a traditional rendering configuration, a single render target is bound with a single depth target. However, this is not a requirement.

The application may bind from 1 to 8 total render targets simultaneously for generating multiple versions of the same rasterized scene. These render targets must match in type (such as Texture2D, Texture2DArray, etc..) and size (including width, height, depth, array size, and sample counts), but may have different formats from one another. When more than one render target is bound, it is referred to as a multiple render target *(MRT)* configuration.

This arrangement also uses a single depth stencil target, even though there are multiple render targets in use. Since the depth stencil target is used to carry out the depth and stencil tests, it must be the same type and size as the resources bound as render targets, but its format will be one of the depth stencil formats. As indicated above, these size-matching requirements hold true for MSAA render targets as well. If an MSAA render target is bound for output, then the depth stencil target must also be an MSAA resource with the same number of subsamples. In addition, the array size of a resource must match between the render and depth stencil targets. You may recall from Chapter 2 that a resource can be created as an array resource, where a number of texture slices are created within the same resource. If a render target has six array slices to produce a cube-map, then the depth stencil target must also have six array slices.

One final configuration is also possible that doesn't use a render target at all. It is possible to bind only a depth stencil view to the pipeline. In this case, each render target slot is emptied by binding a NULL value to it, while the depth stencil view is bound as it is in a regular rendering configuration. This is commonly used to fill the depth stencil target with the depth information of the scene before proceeding with additional rendering passes.

**MRT vs. render target arrays.** A distinction should be noted between how MRTs and render target arrays differ. It is possible to create eight individual texture resources and then bind them to the output merger stage for use in an MRT configuration. It is also possible to create a single texture array resource that has eight texture slices and then bind that resource to the output merger for use in a single render target configuration. Even though they would provide the same number of effective render targets, there are several differences in the mechanics of using these configurations and in their capabilities. An MRT configuration uses more than one render target slot—one for each render target to be used. On the other hand, an array-based resource only occupies a single render target slot in the output merger. This produces another difference between the two configurations—MRT setups write to all of their render targets simultaneously, while array-based setups only write to one slice at a time. Since the pixel shader can write to all MRT render targets simultaneously, only a single pixel shader invocation is needed to write to all targets. Since the pixel shader can only write to a single render target in the array-based configurations, it would take one pixel shader invocation to write to each of its texture slices.

This situation seems to indicate that MRT configurations are a better choice, since this can reduce the number of pixel shader invocations, while still writing to the same

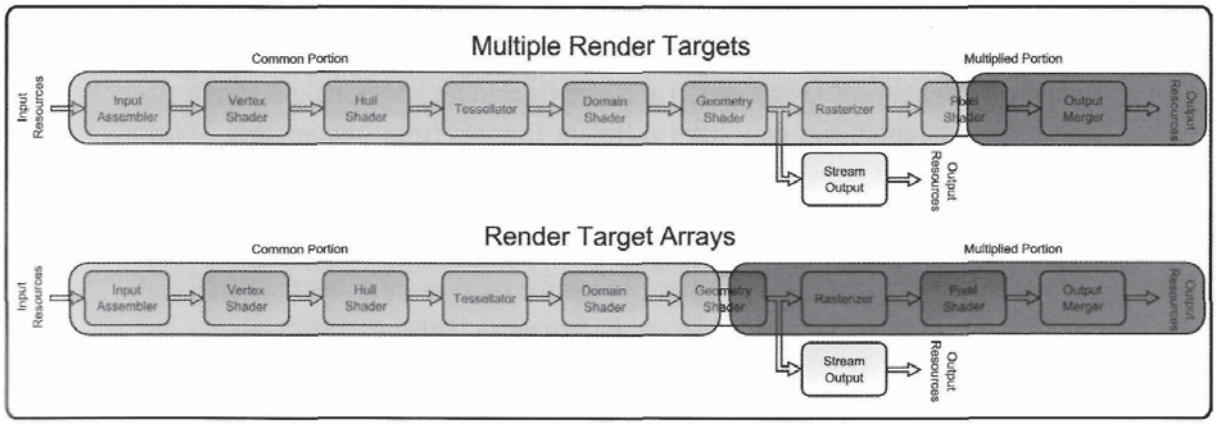



Figure 3.79. The conceptual splitting of the pipeline with MRTs and array-based resources.

number of render targets. However, render target array resources have several advantages, as well. In a sense, we can say that MRT configurations use the same rasterization for all of their render targets. This means that if a triangle is rasterized to the upper-right-hand corner of a render target, then it will appear in the same location for all of the render targets that are written to. Render target arrays each use an individual rasterization that can be customized, based on the SV_RenderTargetArrayIndex system value semantic. This allows different view transformations to be used for the geometry before rasterization, and thus also allows primitives to be rasterized to different locations within the render targets.

We can think of these two configurations as splitting the pipeline to ultimately write to multiple render targets, but they split it in different locations. The MRT configurations split the pipeline at the pixel shader stage, while array-based configurations split the pipeline prior to the rasterizer stage. Both configurations can be useful in different situations, depending on what type of pipeline output is needed. This difference is depicted in Figure 3.79.

There is still another facet to consider regarding the binding of render and depth stencil targets. We mentioned above that these resources are bound with resource views—RTVs and DSVs. Since a subresource of a resource can be specified by a resource view, it is possible to have a larger resource bound to the pipeline through a resource view that works with a smaller portion of that resource. When this is considered along with all of the other options and configurations mentioned above, the developer has a large variety of different options available to create specialized rendering algorithms.

**Binding render targets.** Both the render targets and the depth stencil target are bound to the output merger in a single device context method call. The ID3DllDeviceContext ::OMSetRenderTargets() method (shown in Listing 3.27) takes a pointer to an array of pointers to render target views and a pointer to a depth stencil view in addition to an integer specifying the number of targets in the array. The pipeline will hold a reference to the

render targets after they are bound, and it will keep the reference until the render target is replaced with a NULL reference.

One other resource type can be bound to the output merger stage. We have seen in the section on the pixel shader stage that it can use using unordered access views (UAVs). However, the pixel shader stage cannot accept UAVs for binding. Instead, they are bound to the output merger stage in the same way that render targets are. When UAVs are used, they are bound with the ID3D11DeviceContext::OMSetRenderTargetsAndUnordered AccessViews() method. The first three arguments of this method are identical to the render-target-only version, while the remaining four are used to bind the UAVs. There are a total of eight UAV slots available, and the range of slots affected by this call is selected with the UAVStartSlot and the NumUAVs parameters. The ppUnorderedAccessView parameter is a pointer to an array of UAVs to be bound. Naturally, the number of UAVs in this array must match the number of views specified in the NumUAVs parameter.

The final parameter to this method is another pointer to an array of UINT values. These values provide the current buffer counter value for use in Append / Consume buffers. As described in Chapter 2, buffers used with a UAV contain an internal counter indicating the number of elements present in the buffer. This counter is hidden in the buffer and is maintained by the runtime. However, the value stored in these counters at the time of binding is controlled by what is passed into this array. This lets the application effectively reset the data in the buffer if desired, or if a value of -1 is passed, the current internal buffer counter value is maintained.

The total number of render targets and UAVs being bound must not exceed eight. However, any combination that adds up to eight or less is allowed—including using eight UAVs and no render targets. If there are no render targets bound to the pipeline, the UAVs represent the only outputs for the entire pipeline. Both methods for binding render targets to the output merger are shown in Listing 3.27.

```
void OMSetRenderTargets(
    UINT NumViews,
    ID3D11RenderTargetView **ppRenderTargetViews,
    ID3D11DepthStencilView *pDepthStencilView
);

void OMSetRenderTargetsAndUnorderedAccessViews(
    UINT NumViews,
    ID3D11RenderTargetView **ppRenderTargetViews,
    ID3D11DepthStencilView *ppDepthStencilView,
    UINT UAVStartSlot,
    UINT NumUAVs,
    ID3DIIUnorderedAccessView **ppUnorderedAccessView,
    const UINT *pUAVInitialCounts
);
```

Listing 3.27. The device context methods for binding resources to the output merger stage.

### Read-Only Depth Stencil Views

One other new feature in Direct3D 11 is the ability to use a depth stencil resource as a depth/stencil target in the output merger stage, and to also simultaneously view its contents through a shader resource view in one of the programmable shader stages. At first thought, this would seem to violate the rule that a resource cannot simultaneously be read from and written to from different resource, since the depth stencil resource would be written to in the depth test portion of the output merger.

However, if the depth stencil view is created with the appropriate read-only flags,[37] this ensures that the resource is not written to by the output merger stage. Instead it is only read by the output merger to determine if the depth test has passed or failed (if the depth test is currently enabled). This effectively makes both of the pipeline locations that the resource is bound to read-only access points, which still follows the simultaneous read/write rules. This configuration is depicted in Figure 3.80. This is useful because a secondary rendering pass can use the depth buffer contents without having to copy them into another resource, while at the same time the resource is also being used for the depth test.

## 3.12.3 Output Merger Stage Processing

The output merger stage performs two types of operations—visibility tests and blending operations. The visibility tests provide configurable operations that are used to determine if each particular fragment should be blended into the output resources attached to the output merger. If the fragment passes both the depth test and the stencil test, it is passed to the blending function to be combined with the render targets. The blending function is also configurable, which allows for a wide variety of methods to apply pixel shader outputs to the render target. In this section, we follow the path that a fragment would follow through the output merger stage. We first discuss each of the two visibility tests, to better understand how the tests operate and what modifications can be made to them. Next, we investigate how the blending function performs its duty and examine the available modes for blending.

### Visibility Tests

Two visibility tests are performed simultaneously by the output merger stage for each fragment passed on to it: the depth test and the stencil test. Both of these tests use the depth stencil resource that is bound to the output merger stage with a depth stencil view (DSV), and require an appropriate data format to be selected for the depth stencil resource. The

[37] Two flags are available, D3D11_DSV_READ_0NLY_DEPTH and D3D11_DSV_READ_0NLY_STENCIL, which correspond to the depth and stencil portions of a depth/stencil target, respectively.

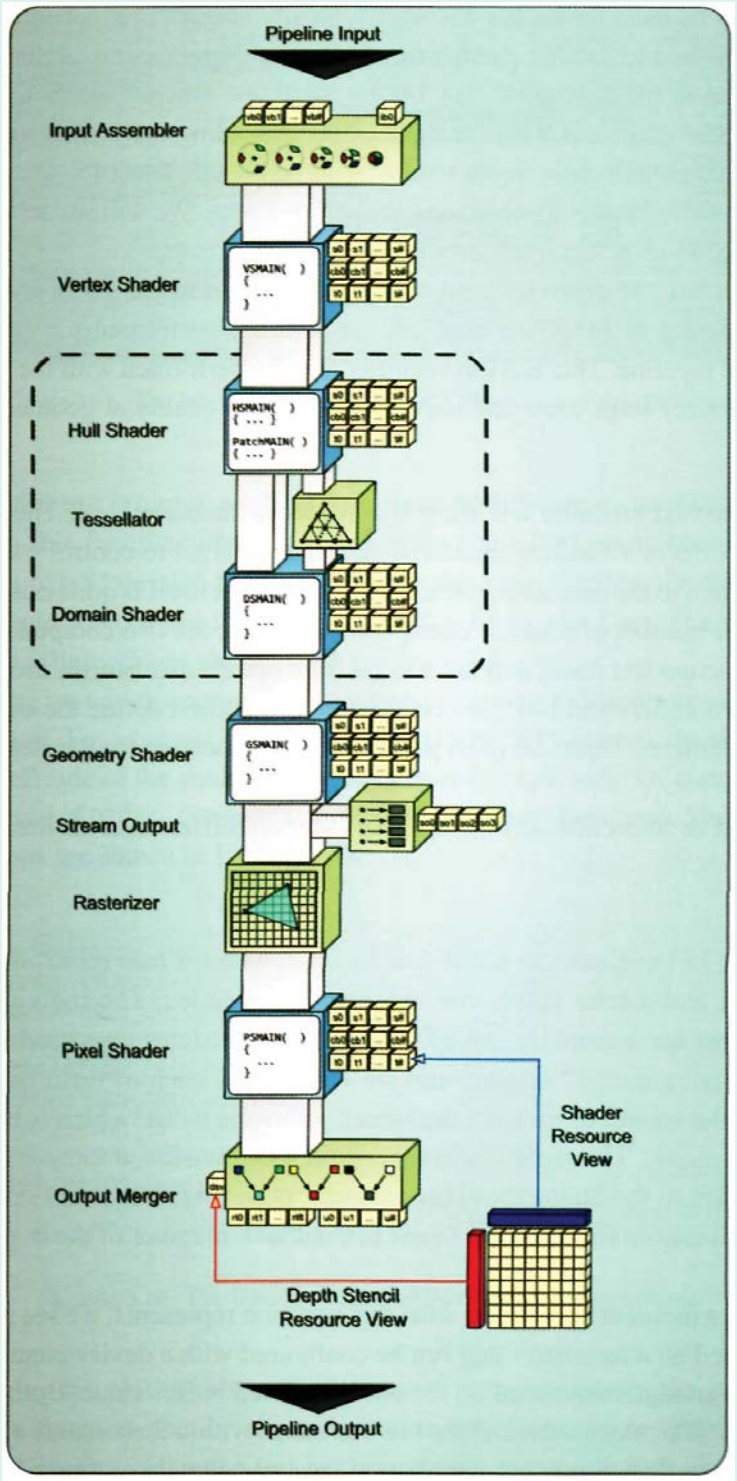



Figure 3.80. A single resource being used in multiple locations of the pipeline, including a depth stencil view in the output merger.

formats that can be used for such a resource typically contain one portion for storing the current depth value, and another portion for storing the current stencil value. One example of such a format is DXGI_FORMAT_D24_UNORM_S8_UINT, where 24 bits are dedicated to holding the depth value, and 8 bits are dedicated to holding the stencil value. There are some formats that contain only depth values, with no data dedicated as a stencil buffer. In these cases, the stencil test is disabled and will always pass. We will see how each of these quantities is used in its respective visibility determination tests.

When enabled, the depth test and stencil test are performed for every fragment sent to the output merger. If MSAA is enabled, the tests are performed for every subsample produced by the pipeline. This lets the visibility tests be performed with the same granularity that the rasterizer stage uses, and improves the image quality at geometry intersection areas.

**Stencil test.** The first visibility test we will consider is the stencil test. This test allows an application to perform a masking operation on a render target to control when a particular fragment is written to the destination render target. The test itself is quite configurable, with a relatively large number of possible configurations. There are two components to this test. The first is the actual test itself, and the second is an update mechanism used to update the stencil buffer. To understand how the test works, we will first define the equation that the stencil test implements. Equation (3.6) provides the pseudocode for the stencil test.

$$(\text{StencilRef} \,\&\, \text{StencilMask})\ \text{CompFunc}\ (\text{StencilBufferValue} \,\&\, \text{StencilMask}) \tag{3.6}$$

Equation (3.6) evaluates to either *true or false,* where a *true* result indicates that the test has passed, and a false result indicates that it has failed. The individual arguments of the stencil test are a combination of the configurable states discussed in the "Output Merger State Configuration" section, and we will review each of them here. Starting on the left side of the equation, we have the stencil reference value, which is bitwise-ANDed with the stencil mask. The stencil reference value is an unsigned integer value provided by the application in the ID3DllDeviceContext::OMSetDepthStencilState() method, while the stencil masks are set as the StencilReadMask member of the depth stencil state description.

If we take a moment to consider what this equation represents, we see an argument on the left side based on a parameter that can be configured with a device context method. On the right side is an argument based on the current stencil buffer value. Both arguments are masked to allow selecting a subset of the bits contained within their source variables. These two arguments are then compared with one of several comparison functions provided by the API. Thus, all three major portions of this equation are configurable, allowing for a wide variety of possible tests.

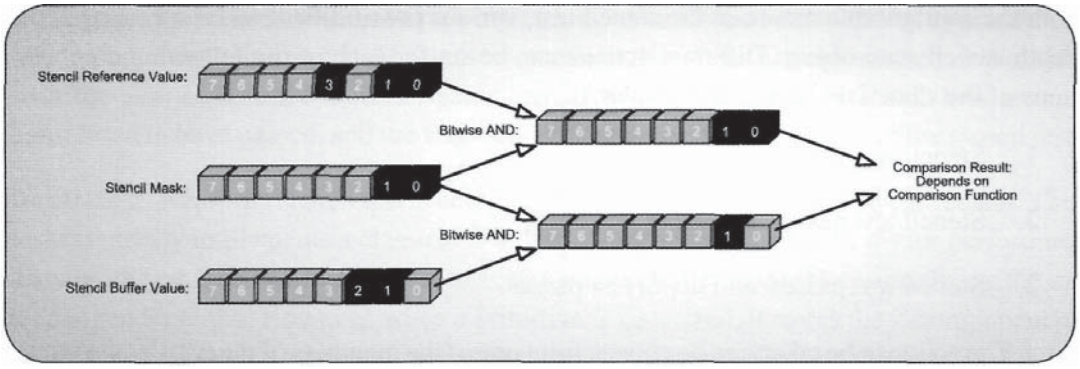



Figure 3.81. The bit registers for each of the arguments of the stencil test.

Because this test is highly configurable, it can be difficult to visualize how it operates. To help clarify this functionality, we will provide a small example scenario for a single fragment being passed through the stencil test. In this case, the stencil reference value will have bits 0, 1, and 3 set, while the stencil buffer has bits 1 and 2 set. The stencil mask has bits 0 and 1 set. These values are depicted in Figure 3.81.

Depending on which comparison function is chosen, a different result can be found for the stencil test. For example, if D3D11_COMPARISON_LESS is used, the stencil test would fail since the left side of the equation is greater than the right side. Of course, this function could be reversed if D3D11_COMPARISON_GREATER were used instead. The available comparison functions are shown in Listing 3.28.

```
enum D3DII_COMPARISON_FUNC {
    D3D11_COMPARISON_NEVER,
    D3D11_COMPARISON_LESS,
    D3DII_COMPARISON_EQUAL,
    D3DII_COMPARISON_LESS_EQUAL,
    D3DII_COMPARISON_GREATER,
    D3DII_COMPARISON_NOT_EQUAL,
    D3D11_COMPARISON_GREATER_EQUAL,
    D3D11_COMPARISON_ALWAYS
}
```

Listing 3.28. The D3D11_COMPARISON_FUNC enumeration.

Considering the ability to specify the stencil reference value, the stencil mask, and the comparison function, the developer has more or less complete control over how the stencil test will operate. After the test has completed, some action needs to be taken, depending on if the test has passed or failed. In fact, the action taken relies on both the stencil test result and the depth test result (which we will discuss in the next section). In keeping

with the configurable nature of the stencil test, various possibilities can be specified in the depth stencil state object. Different actions can be set for each of the following combinations of the stencil and depth test results:

1. Stencil test fails.
2. Stencil test passes, but depth test fails.
3. Stencil test passes, and depth test passes.

The action to be taken can be chosen from one of the members of the D3D11_STENCIL_OP enumeration, which is shown in Listing 3.29.

```
enum D3D11_STENCIL_0P {
    D3D11_STENCIL_0P_KEEP,
    D3D11_STENCIL_0P_ZER0,
    D3D11_STENCIL_0P_REPLACE,
    D3DII_STENCIL_OP_INCR_SAT,
    D3D11_STENCIL_0P_DECR_SAT,
    D3D11_STENCIL_0P_INVERT,
    D3D11_STENCIL_0P_INCR,
    D3DII_STENCIL_OP_DECR
}
```

Listing 3.29. The D3D11_STENCIL_0P enumeration.

These options determine what is done with the stencil buffer value after the tests complete. There are options for keeping the existing stencil buffer value; clearing it to zero; replacing it with the stencil reference value; and incrementing, decrementing, or inverting it. Once again, these options provide significant freedom in using the stencil buffer for a variety of different algorithms, since different actions can be taken, depending on the test results. One final operation is performed before the stencil value is written to the stencil buffer. The value produced by the stencil operation is bitwise AND'ed with the StencilWriteMask before being stored.

In this section, we have seen the configuration of a complete stencil test. However, there are some additional configurations to discuss. All of the settings shown above can be individually configured for front faces and back faces with the FrontFace and BackFace members of the depth stencil state object. These members are actually structures that encapsulate one set of stencil test configurations (both the comparison function and the operations to perform in each test result case). This lets special tests to be implemented that perform different options for front and back faces. The most common example of using different settings

in this manner is the shadow volumes algorithm.[38] In addition to the ability to use different configurations for front and back faces, the overall test can also be enabled or disabled with the StencilEnable Boolean parameter. If the stencil test is disabled, it is always considered to have passed, and the fragment will never be culled because of the stencil test.

Depth test. While the stencil test is being performed, the depth test is also carried out. This test essentially implements a classical Z-buffer algorithm (Williams, 1978) for performing a visibility test. The basic concept is to keep a second buffer that is the same size as the intended render target. However, when a primitive is rasterized, it stores the Z-component of the normalized device coordinates of each fragment, instead of storing the rasterized color values in the buffer. This produces a buffer that contains the post-divide Z-component of the fragment position with values in the range of [0.0,1.0]. In the output merger stage, this Z-buffer is implemented as the depth portion of the depth stencil buffer, and is typically referred to as a *depth buffer,* since the Z-component represents a measure of the distance from the viewer.

When these depth values are stored, any additional primitives that are rasterized to create fragments can compare their own depth value with the one stored in the depth buffer. If the depth value stored in the buffer is smaller (closer to the viewer) than the value for the new fragment, then the new fragment can be discarded, since it is located behind another object in the scene. If the stored depth buffer value is larger than the new fragment (farther from the viewer), then the new fragment's color value is passed to the rest of the pipeline, and the Z-buffer value is updated to represent the new visibility information. In this way, the Z-buffer provides per-pixel (or per-sample if MSAA is used) visibility determination.

The output merger implements the Z-buffer algorithm with several additional configurations for preparing the depth value, as well as actually performing the depth comparison and writing the result to the Z-buffer. These configurations are primarily contained within the depth stencil state object, with the exception of the viewport. We will follow a fragment through the depth test in the same fashion as we have done for the stencil test, to gain a better understanding of how the process works.

The depth test can be enabled or disabled with the DepthEnable member of the depth stencil state object. This parameter only determines if the depth test is performed; it does not control if the depth writing functionality is enabled or disabled. The depth value is received either directly from the rasterizer stage, or if the pixel shader program modifies the depth value then it is received from the pixel shader stage. The value is then clamped to the min and max depth values specified in the viewport structure that was used to generate the fragment. This clamping is performed in a depth buffer format appropriate way. After the depth range is clamped, the depth value is read from the depth stencil buffer, and the two values are compared with a selectable depth-comparison function, selected

[38] This algorithm was first described in the "Rasterizer" section of this chapter.

with the DepthFunc member of the depth stencil state object. Listing 3.30 provides all of the available comparison functions.

```
enum D3D11_C0MPARIS0N_FUNC {
    D3D11_C0MPARIS0N_NEVER,
    D3D11_COMPARISON_LESS,
    D3D11_COMPARISON_EQUAL,
    D3D11_C0MPARIS0N_LESS_EQUAL,
    D3D11_C0MPARIS0N_GREATER,
    D3D11_C0MPARIS0N_N0T_EQUAL,
    D3D11_C0MPARIS0N_GREATER_EQUAL,
    D3D11_C0MPARIS0N_ALWAYS
}
```

Listing 3.30. The D3D11_COMPARISON_FUNC enumeration.

These comparisons are performed with the fragment depth value on the left side, and the depth buffer value on the right side. If the comparison evaluates to *true,* then the depth test has passed, and the fragment continues on in the process. If the comparison evaluates to *false,* then the depth test is concluded, and the fragment is discarded. For example, the standard comparison is to use D3Dll_COMPARISON_LESS, which provides functionality similar to the original Z-buffer algorithm. If the fragment depth is less than the depth buffer value, it will pass the depth buffer test. This indicates that the new fragment is closer to the viewer, and should be visible.

If the depth test fails, the fragment is discarded. If it passes, it will be passed to the blending functionality (which will be covered in the next section). The depth buffer may be updated, depending on a few conditions. If both the depth test and the stencil test have passed, the fate of the depth value depends on whether depth writes are enabled or not. This is specified with the DepthWriteMask member of the depth stencil state. If the value is set to D3D11_DEPTH_WRITE_MASK_ALL, then the depth value can be updated in the depth buffer; otherwise, the depth value is discarded.

It may seem counterintuitive that the depth buffer could be used for depth testing, but still have a configuration that allows it to not be written to. In fact, there are many situations where this is the preferred behavior in modern rendering schemes. It is quite common for a first rendering pass to be used to fill the depth buffer with values using the standard Z-buffer technique described above. Then, any subsequent passes would use the depth buffer as it is, since all of the geometry has been rasterized before. In effect, the visibility has already been determined, so there is no need to have depth writing enabled. A performance benefit is available when the depth writing is disabled, since the depth buffer would only need to be read from and never written to.

This "read-only depth buffer" concept can be taken a step further. You may recall that in Chapter 2 we saw that a depth stencil view could be created with flags to indicate that

either the depth component or the stencil component, or both, can be created as read-only. This means that the view itself ensures that the resource can only be read from, which allows it to be used in more than one location in the pipeline. The resource can be bound for use in the depth test and simultaneously used as a shader resource. This can be very beneficial when the depth buffer is used as an input to later rendering passes, but still needs to be used for the depth test at the same time.

## Blending

If a fragment has survived both the stencil test and the depth test, it is passed to the blending function. The blending function makes it possible to combine two selectable values prior to writing them to the output render target, and the function used to combine the values is also selectable. This functionality has traditionally been used to perform alpha-blending to implement partially transparent rendering materials. However, with the large number of possible blend sources, operations, and write options, there is plenty of additional functionality that can be used for other, less conventional techniques. The blending function is controlled by the configurations contained within the blend state object. Listing 3.31 provides the list members that can be manipulated in the blend state.

```
Struct D3D11_BLEND_DESC {
    BOOL                                AlphaToCoverageEnable;
    BOOL                                IndependentBlendEnable;
    D3D11_RENDER_TARGET_BLEND_DESC      RenderTarget[8];
}
```

Listing 3.31. The D3D11_BLEND_DESC structure.

Here we find two top level configurations, followed by an array of eight render target blend description structures. The first top-level member is the AlphaToCoverageEnable parameter, which we have already discussed in the "Pixel Shader" section of this chapter. The second top-level member is another Boolean value, which determines if all of the render targets bound for output in the output merger stage will be blended independently, or if they will all use the same blending configuration. As you may have guessed by now, the blending configurations are stored in the eight-element array member of the blend state, since there is a maximum of eight render target slots available. If individual blending is enabled by setting IndependentBlendEnable to *true*, each of these array elements defines the blending mode to use for the corresponding render target slot. If independent blending is disabled, then all render targets will use the blending configuration from index 0 of the array. If it is not needed, then independent blending should be disabled to allow for better performance.

To gain insight into how the blending configuration is used, we will use the same paradigm that we employed for the stencil and depth tests and follow the path of a fragment that is running through the blending functionality. Listing 3.32 shows the members of the D3D11_RENDER_TARGET_BLEND_DESC structure.

```
Struct D3D11_RENDER_TARGET_BLEND_DESC {
    BOOL            BlendEnable;
    D3D11_BLEND     SrcBlend;
    D3D11_BLEND     DestBlend;
    D3D11_BLEND_0P  BlendOp;
    D3D11_BI_END    SrcBlendAlpha;
    D3D11_BLEND     DestBlendAlpha;
    D3Dll_BLEND_OP  BlendOpAlpha;
    UINT8           RenderTargetWriteMask;
```

Listing 3.32. The D3D11_RENDER_TARGET_BLEND_DESC structure.

The first member, BlendEnable, performs exactly what its name states. It enables or disables the blending functionality. We will assume that blending is enabled for this example. Next, we see two sets of three states. The three members following BlendEnable are used to blend color values (the RGB components of the fragment color), and the next three are used for blending the alpha value (the A component of the fragment color). This separation of blending state allows completely different blending modes to be used for alpha and color, which can let them be used for separate purposes.

The three members that are supplied for both color and alpha are used to define the blending equation. The two data sources for the blending equation are selected by SrcBlend and DestBlend parameters for the color blending, and SrcBlendAlpha and DestBlendAlpha for alpha blending. These parameters define their data source as one of the members of the D3Dll_BLEND enumeration, which is shown in Listing 3.33.

```
enum D3D11_BLEND {
    D3D11_BLEND_ZER0,
    D3D11_BLEND_0NE,
    D3D11_BLEND_SRC_C0L0R,
    D3D11_BLEND_INV_SRC_C0L0R,
    D3D11_BLEND_SRC_ALPHA,
    D3D11_BLEND_INV_SRC_ALPHA,
    D3D11_BLEND_DEST_ALPHA,
    D3D11_BLEND_INV_DEST_ALPHA,
    D3D11_BLEND_DEST_C0L0R,
    D3D11_BLEND_INV_DEST_C0L0R,
    D3D11_BLEND_SRC_ALPHA_SAT,
    D3D11_BLEND_BLEND_FACT0R,
    D3D11_BLEND_INV_BLEND_FACT0R,
```

```
        D3D11_BLEND_SRC1_C0L0R,
        D3D11_BLEND_INV_SRC1_COLOR,
        D3D11_BLEND_SRC1_ALPHA,
        D3D11_BLEND_INV_SRC1_ALPHA
}
```

Listing 3.33. The D3D11_BLEND enumeration.

As you can see, there are a substantial number of possible data sources. The general options in this list include data sources such constant values and color/alpha values, with several possible modifiers. The color and alpha sources all provide source and destination modifiers, where the source is the value from the current fragment and the destination is the value located in the "destination" render target. Each of these values also provides an "inverse" value, which is generated by using 1 minus the value. For example, the D3D11_BLEND_SRC_C0L0R selects the current fragment color as its source. However, D3D11_BLEND_INV_DEST_C0L0R selects the inverse of the destination render target value as its source. In addition to each of these modified color and alpha values, it is also possible to use one of several constant values. The constant values that can be chosen are either 0, 1, or the blend factor that is set by the application with the ID3DllDeviceContext: :0MSet BlendState() method. The blend factor also provides an inverse value as well.

You may also notice that there are several parameters with the SRC1 modifier in their names, such as D3D11_BLEND_SRC1_C0L0R and D3D11_BLEND_INV_SRC1_ALPHA. These values are actually taken from the second output register in the pixel shader, and can be used in blending operations at the same time as the first output register. This lets two source quantities be read from the current fragment, instead of using the contents of the destination buffer. Using both of the output register values is referred to as dual-source color blending, and can only be used with a single render target output (otherwise, the second color value would be passed to the second render target).

Once the data sources have been selected, the blending operation must be selected. This operation combines the two data source values into a single value to be written into the output render target destination. The available operations are provided in Listing 3.34. The available operations are fairly self-explanatory and implement what their names imply. There are three arithmetic operations (A+B, A-B, B-A), along with minimum and maximum selection options.

```
enum D3D11_BLEND_0P {
      D3D11_BLEND_0P_ADD,
      D3D11_BLEND_0P_SUBTRACT,
      D3D11_BLEI\ID_0P_REV_SUBTRACT,
      D3D11_BLEND_0P_MIN,
      D3D11_BLEND_0P_MAX
}
```

Listing 3.34. The D3D11 BLEND OP enumeration.

As previously mentioned, these blending operations can be performed separately for color values and alpha values. Once the blending operation has completed, a complete four-component RGBA color is available for writing. The final step in the blending process is to use the render target write mask to determine which channels of the output color will actually be written. This is specified in the RenderTargetWriteMask member of the blend state structure. The RGBA channels correspond to the bit positions 0, 1,2, and 3 of the mask. When the bit is set, the channel is written to the render target.

### 3.12.4 Output Merger Pipeline Output

We have seen throughout this section what operations are performed on the pipeline data before it is written to the resources bound to the output merger stage. There are a total of three different types of resources that can be written to by a pipeline execution: render targets, unordered access views, and the depth stencil target.

The render targets receive the output from the blending functions. Each fragment written to a render target must have survived both the stencil test and the depth test before being blended and finally written to a render target. If MRTs are being used, then each render target will have a blended value written to it that originates from the pixel shader. If render target arrays are used, then the appropriate render target slice is determined back at the input to the rasterizer stage with the SV_RenderTargetArrayIndex system value semantic. Thus, by the time the data reaches the output of the blending function, the appropriate texture slice has already been selected.

In contrast, data written to a UAV is directly controlled by the pixel shader stage and does not need to pass any tests to successfully make it to the output resource. When the pixel shader modifies a resource, those changes are immediately submitted to the GPU and take effect as soon as the GPU's memory system writes the values to memory. There is some delay in this writing process, and hence, its results must not be assumed to be immediately available when designing an algorithm.

The depth stencil target is somewhat more complex. Output data must pass the depth test and the stencil test to be written to the depth portion of the output resource. In addition, depth writing must be enabled in the depth state object. Stencil data can be updated differently, depending on which combination of the depth and stencil tests pass. Since these are configurable settings, it is more or less up to the application to determine when and how the stencil data is updated by the data stream coming into the output merger stage.

After completion of the pipeline execution, the modified contents of each of these resources are available for use in further rendering passes and computation passes, or even for direct manipulation by the CPU. In the case of MSAA rendering, one further step is needed before using the resource. After the rendering is finished, it is typically desirable to resolve the MSAA render target to a non-MSAA texture. In Direct3D 11 this is done

by calling the ID3DllDeviceContext: :ResolveSubresource() method. Performing this resolve combines the values of the individual subsamples into a single value, resulting in a texture suitable for being displayed on the screen, or for being sampled using normal shader sampling methods. The filter applied when performing the resolve is specific to the hardware and driver, and to the quality level specified for the render target. However, in most cases, a box filter is used. For a box filter, all subsamples are weighted equally, making it a simple arithmetic average of the subsample values.

## 3.13 High-Level Pipeline Functions

After stepping through the entire pipeline, stage by stage, we have covered a significant amount of functionality in great detail. In fact, there is so much information presented in this chapter that it can be overwhelming to consider all of the different types of calculations that can be performed. Therefore, we will provide a few ways to consider the functions of each stage from a somewhat higher level, in an attempt to clarify how they are typically used. These groupings are generalized based on typical usage, but they don't necessarily reflect any type of requirement. After all, the pipeline's flexibility allows for very creative and unusual uses, so these are merely presented as a starting point for the reader to build from.

### 3.13.1 Vertex Manipulations

The early stages of the pipeline are generally used to build and manipulate the vertices that represent the geometric surfaces that will be rendered. This includes construction of vertices by the input assembler, as well as the individual per-vertex operations that are performed by the vertex shader. At the lowest level, these two stages perform the most basic operations on the input geometry that are seen throughout the pipeline. Because of this close proximity to the original input geometry, it is quite natural to make modifications to the geometry early in the pipeline. Traditionally, the transformation matrices are applied in the vertex shader, which is essentially manipulating the geometric properties of the model.

In general, since both of these stages operate at a per-vertex level, they will most likely be executed the least during a single pipeline execution. For each draw call, the input assembler will produce each vertex of the model, and will set up each primitive for further processing later in the pipeline. From these two streams, the vertex shader will operate only once per vertex. In contrast to the later stages in the pipeline, the input assembler and vertex shader perform their calculations at a relatively low frequency. While the input assembler is a fixed function stage, the vertex shader is programmable and can be

used to implement a customized routine. This customizable processing can be used to our advantage by performing calculations in the vertex shader that can be passed on to the later stages, instead of calculating everything directly in the later stages.

It is also logical to perform any other calculations in the vertex shader that provide relatively low-frequency information to the surface of the geometry. For example, it is often adequate to calculate ambient lighting calculations in the vertex shader, since it doesn't change very quickly across the entire model. By performing the calculation at the vertex level, we relieve a small amount of calculation from each shader invocation later in the pipeline.

### 3.13.2 Tessellation Manipulations

The next set of pipeline stages implements the tessellation functionality of the rendering pipeline. The hull shader, tessellator, and domain shader are all used together to provide a very flexible and robust tessellation system. This can be seen as an extension of the previous vertex-based processes, since it ultimately produces vertices, as well. In many ways, the domain shader can be seen as a repeat of the vertex shader, with some additional responsibilities for creating the vertex before it is processed.

However, the tessellation stage is intended to amplify the available detail at a geometric level, which is a departure from what the input assembler and vertex shader can do. This means that the overall number of invocations of the domain shader will be significantly higher than it is for the vertex shader. Because of this, and because of the additional calculations that need to be performed in the domain shader, the vertex shader should be used as much as possible, instead of the domain shader. If all things are equal, the vertex shader should be chosen over the domain shader.

Even so, the tessellation system provides some unique opportunities for the developer. By amplifying how many vertices are available to represent the geometry, we can perform much higher frequency calculations much earlier in the pipeline than was possible in previous versions of Direct3D. In the extreme case, we can produce one vertex for every pixel, making the visual fidelity of the per-vertex data equal to that of the per-pixel data. We can use this to our advantage by moving calculations to earlier in the pipeline and away from the raster based stages, or we can simplify the later stages by performing more complex calculations at the vertex level.

### 3.13.3 Geometry Manipulations

The geometry shader and the stream output stages are very much specialized stages that sit between the vertex-based stages and the raster-based stages. As such, they are intended

to perform higher level geometric operations than the vertex based stages do, before the geometry is rasterized. While this is extremely useful in some situations, it is less common to use the geometry shader in most algorithms. The majority of the algorithms that use geometry shaders use them because of its special features that aren't available in any other stage. The expansion of points into quads is a good example of functionality that can't be performed in any of the other stages.

This is probably at least partially due to the poor performance that the geometry shader became known for during its debut in Direct3D 10, and to a corresponding lack of development effort geared toward it. However, with the shared processor architectures that most current generation GPUs use, sufficient processing power is available to use the geometry shader. The biggest challenge is to ensure that the memory usage of the stage is appropriately balanced with the tasks that are being performed. For example, if each geometry shader invocation is used to produce 100 output triangles, the algorithm should likely be reevaluated to take advantage of the tessellation stages, instead of the geometry shader. However, for small-scale geometry manipulation, there is a much better balance of memory usage to computation. This makes point sprite expansion an attractive target—it performs some calculations, and a small amount of data amplification. At the very least, it will be interesting to see if more algorithms are developed to use the geometry shader in the coming years, or if it will remain as a specialty stage in the pipeline.

### 3.13.4 Raster-Based Manipulations

The final portion of the pipeline is the raster-based stages. These include the rasterizer, pixel shader, and output merger. These stages operate at a much higher frequency that the previous stages, due to the data amplification that is performed in the rasterizer. This allows for much higher frequency operations to be performed in these stages, which is why the pixel shader is so frequently used to add all of the detail to a rendering. The rasterizer is typically not a bottleneck in rendering algorithms, while the pixel shader can potentially be either a calculation or memory bandwidth bottleneck, due to the large number of invocations that are performed.

A somewhat less recognized performance issue can be introduced by the output merger. Since it reads and writes to the depth stencil buffer and can potentially read and write to the render targets when blending is enabled, the output merger can greatly increase bandwidth usage. If you don't need to use the blending function, make sure it is disabled! Likewise, if there is no reason to expect updating of the depth buffer (such as in a second or third rendering pass), ensure that the depth writing functionality is disabled.

In addition to these considerations, an entire class of algorithms is just now starting to be developed that use the unordered access views in the pixel shader. The ability to perform custom reading and writing of resources at arbitrary locations provides a totally new way to

fill resources with data, from a location that was previously unavailable. Some totally new demos are being produced by the GPU manufacturers that efficiently use this new ability in very interesting ways. As we further explore this functionality, we will see how developers can manipulate it for either a performance or image quality improvement.