4 The Tessellation Pipeline

4.1 Introduction

Tessellation is one of the headline new features for Direct3D 11. While version 11 brings many improvements, only a few are wholly new compared with others that are incremental revisions on previous releases.

Tessellation not only introduces two new programmable units and one fixed-function, all of which require new knowledge and expertise from the developer; it also opens up new opportunities for artists and content creators. It can revolutionize how real-time computer graphics operates.

This chapter will take the concepts introduced earlier in Chapter 1 and Chapter 3, starting by providing a detailed discussion of the motivation and concepts behind the tessellation feature. The chapter concludes by covering numerous parameters and entry points exposed to the developer.

Chapter 9 will cover examples of using this technology.

4.1.1 What Is Tessellation?

The origins of the word *tessellate* date back to Latin, in the context of mosaics. The word *tessella* refers to the small fragments used to create a larger mosaic, while *tessellation* (or tiling) means to cover a surface, with no gaps between fragments, and no overlaps.

This can be extended to computer graphics in the general sense of using many smaller pieces of geometry to create, without gaps or overlaps, a larger complete surface. At its simplest, a common closed mesh of triangles satisfies this definition.

A number of other related terms are used within the field of computer graphics to formalize tessellation as a means of representing non-linear mathematical surfaces. Triangles are typically the preferred unit of raster-based computer graphics, due to their very useful properties of being convex and having coplanar vertices, but this convenience also limits them to representing flat geometric surfaces. Many small triangles may be used to approximate a smooth surface, but the basic unit of the triangle is still fiat.

Two of the better known techniques for representing a curved surface are *non-uniform rational basis splines (NURBS*) and *SubDivision surfaces* (often abbreviated as *SubDs).* The former is a generalized mathematical form (Weisstein] for smooth, curved surfaces; it has been a staple feature in graphics software for decades. The latter is a general framework for mesh refinement, recursively adding triangles until it better represents the ideal surface. *Catmull-Clark subdivision surfaces* (Catmull & Clark, 1978) are a commonly used example. The crucial difference is that subdivision surfaces don't require a mathematical basis, whereas using NURBS does.

Mathematically defined surfaces are often referred to as *higher-order surfaces* because their underlying equations are defined in terms of their order—quadratic and cubic are most common—and because this order is above linear.

Various tessellation algorithms allow developers or artists to create idealized surface of curves and smooth surfaces to be mapped to conventional triangle-based raster hardware. The individual triangles are the small stone fragments of the final curved-surface mosaic.

4.1.2 Why Is Tessellation Useful?

As previously discussed, a large number of small triangles can be used to approximate a higher-order smooth surface. It is therefore useful to understand why we need additional complexity and hardware when we could simply stick with existing techniques.

This new technology available in Direct3D 11 solves real technical, artistic, and business problems in the domain of real-time computer graphics.

The demand for increasing image quality imposes a significant strain by greatly increasing the volume of data required to define high-resolution models. This increased volume of data requires a correspondingly large increase in on-disk and in-memory storage, I/O bandwidth, and number of calculations. A mathematical definition for a surface requires storage of only the coefficients or inputs into the appropriate function, much less than needed to store raw triangles. Being able to dynamically scale output based on fixed inputs also offers convenience for developers looking to scale image quality across multiple grades of hardware, thus reducing problems involved in targeting different hardware configurations.

Despite these immediately obvious savings in storage and bandwidth, it is important to note the trade-off. Functions that define higher-order surfaces must be sampled in real time and cannot be precalculated. These functions can be mathematically complex and require non-trivial processing time by the GPU. On the whole, GPU tessellation pays for itself, but it doesn't come for free.

Higher-order surfaces have been a common tool in content creation packages for decades and are well understand by any experienced artist. Previously, artists had to work only with triangle-based meshes, which complicated their workflow by requiring them to participate in the implementation details, as opposed to the purely artistic and conceptual.

Tessellation is therefore useful in both technical and artistic contexts. These advantages indirectly also benefit businesses, by reducing the amount of time and effort required.

A technical solution that uses resources more conservatively and scales well across different GPUs can allow for less complex code, having a single code path that can be applied to many configurations. One historical difficulty with real-time graphics has been the common need for many code paths to be implemented and fine tuned for each major hardware configuration, which quickly becomes a maintenance and development nightmare. Cleaner development can only be a good thing.

Enabling artists to be more productive greatly reduces the time needed to produce high-quality art assets. Regarding content creation, it has been noted that the drive for higher-quality real-time graphics has put more strain on art production than it has software. Direct3D 11 helps balance this.

4.1.3 History of Tessellation

Curved surfaces and tessellation algorithms are not new concepts. Traditionally, they have only been available to offline rendering, since their complexity was prohibitively high for real-time applications. A well-known example is films using computer-generated imagery *(CGI)* from Pixar (such as *Toy Story)* and DreamWorks (such as *Shrek).* Outside of media, design and engineering were common early applications of this style of mathematics and rendering. In 1962 Pierre Bezier popularized Paul de Casteljau's work on what has become known as a *Bezier curve* (or *Bezier surface)* while working on car designs for Renault. The advantages of using mathematical curves for design became immediately obvious, for many of the reasons discussed above.

Over the decades, various tools, often for computer-aided design *(CAD),* were developed to allow manipulation of higher-order surfaces. For higher-order surfaces to be combined with real-time rendering, as in the early days of commodity three-dimensional computer games, it was necessary to "bake" the mathematical surface into a fixed triangle equivalent that could be used by the rendering software and hardware of the day. This made it impossible to scale the quality and resolution of surfaces, since the required information had been lost.

In the context of computer games, it was ID Software's id Tech 3 graphics engine, used in *Quake 3 Arena* in December 1999, that made waves with regard to using curved surfaces for real-time graphics.

In late 2001, ATI Technologies introduced the Radeon 8500 GPU, which included its TruForm functionality (ATI, 2001). This was a first for commodity hardware, which, when combined with Direct3D 8.1, allowed for hardware-accelerated tessellation of higher-order surfaces.

The popularity and use of this pioneering new feature were short-lived, with the advent of the id Tech 4 engine (culminating in August, 2004's *Doom 3)* and real-time stencil shadow techniques. *Shadow volumes* were a geometric technique that required processing on the CPU for accurate results; this conflicted with TruForm, in which the finally rendered geometry was different from that available to the CPU. Visible differences between the silhouette of a model and its shadow made them incompatible. Ultimately, shadowing won the popularity contest.

Interestingly, around the same time, Valve Software were working on its Source Engine (culminating in *Counterstrike: Source* and *Half Life 2,* both released in summer, 2004). The Source Engine supported shadow mapping (Valve Software) as an equivalent feature to id Tech 4's stencil shadows; in particular, this technique was compatible with tessellated geometry.

Competition between hardware vendors and the two rival software engines was intense. Ultimately, TruForm lost out, due to the popularity of stencil shadows and a lack of universal support across all hardware vendors.

Following on from ID Software's id Tech 4, it became practical and desirable to simulate higher-resolution models by using low-density meshes and texture trickery. Tangent-space normal mapping and related environment mapping techniques can represent the lighting interactions of a geometrically complex surface, while still being texture-mapped to a far simpler planar model. Although this is generally considered to be a reasonable tradeoff between performance and image quality, it had a significant drawback, in that a model's silhouette was still low-detail, which showed up the trick for what it truly was.

4.2 Tessellation and the Direct3D Pipeline

At first glance, the new tessellation stages that have been added in Direct3D 11 don't appear overly complex. But on closer inspection, we can see that when one is designing and writing code, the flow of data and the responsibilities of each unit can quickly become confusing. This is compounded because the deeper pipeline is harder to visualize—a classic pipeline consisting of a vertex shader and pixel shader was relatively simple, and small

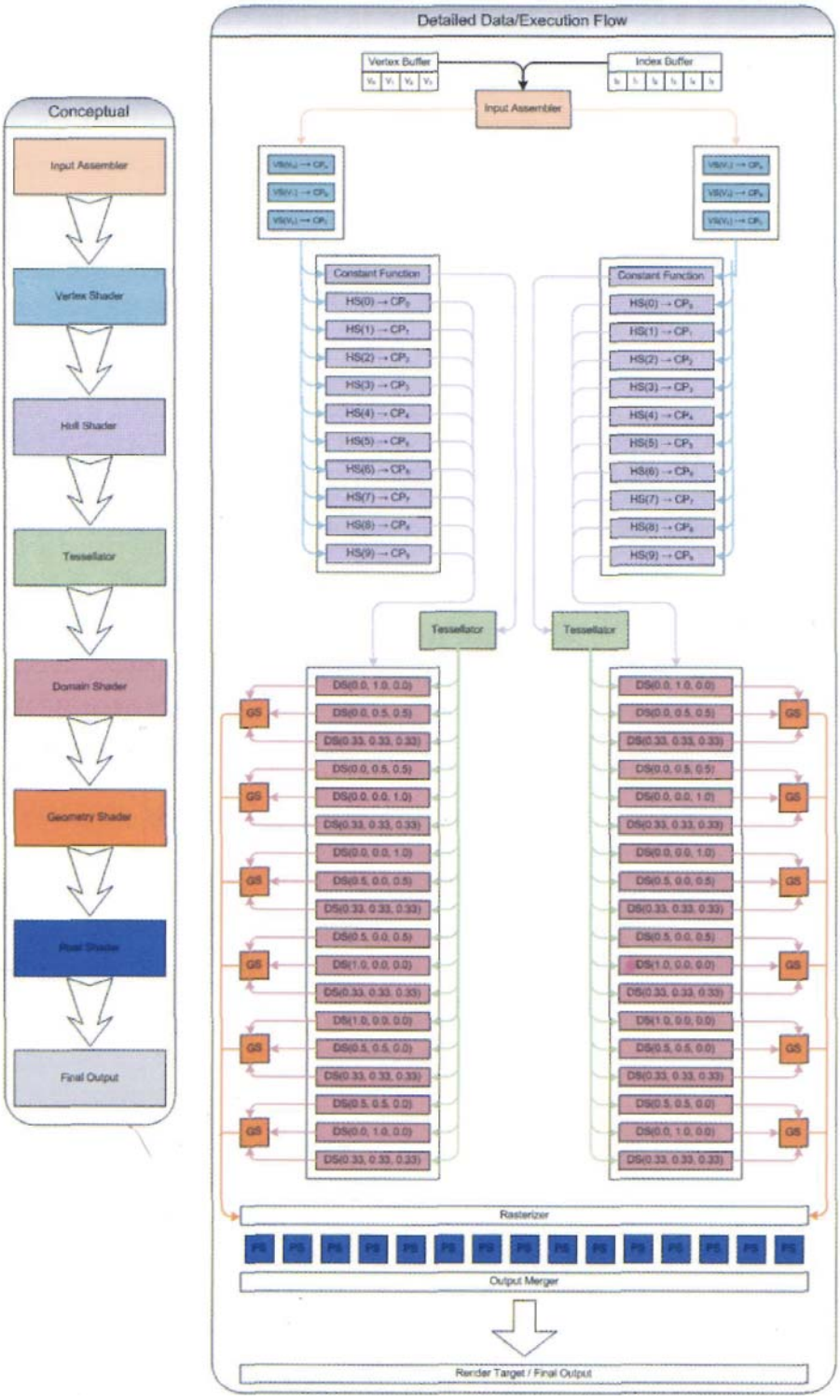

Figure 4.1. Pipeline flow with tessellation enabled.

enough to be held in a developers head. But trying to juggle all six programmable stages can be much more difficult!

The remainder of this chapter goes into more detail on the concepts and general pipeline issues introduced as part of Chapter 3. A knowledge of the higher-level concepts of the pipeline, resources, and shaders is assumed from here onward, so these concepts will not be explicitly described again.

Figure 4.1 can aid in visualizing what is happening at various stages of the new rendering pipeline, by mapping the logical pipeline states to the operations they perform. The precise algorithm is unimportant for the diagram, because the idea of tessellation, which will be described in more detail later, is relatively abstract from a pipeline execution perspective.

To help make Figure 4.1 clearer, the conceptual view discussed in the previous section has been included with a matching color scheme. Arrows show the flow of data and/or execution, and individual shader functions are denoted in the form *xx(yy)* —> *zz,* where *xx* is the abbreviation of the shader type,*yy* represents important parameters, and *zz* is the output.

Figure 4.1 assumes the naive approach of executing a stage as many times as there are outputs or inputs. It is expected that hardware can take advantage of commonality (using pre- or post-transform caches, for example) and reduce the number of invocations. The two best candidates are the vertex and domain shaders; in this example there are 6 VS invocations and 36 DS invocations, yet there are only 4 unique control points and 14 domain points. Specifically, the example used here would do 50% more vertex shading and 157% more domain shading. In a field where performance is crucial, it's easy to see why the hardware would want to be more efficient!

In this example there are four vertices defining a quad on the XZ plane and six indices defining two triangles. Unlike in many other tessellation algorithms, no adjacency information is required, so this doesn't appear to be any different from rendering a normal quad. Without tessellation, the output would like Figure 4.2.

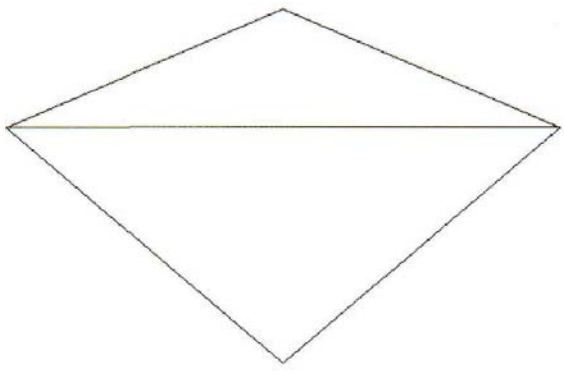

Figure 4.2. Quad rendered without tessellation.

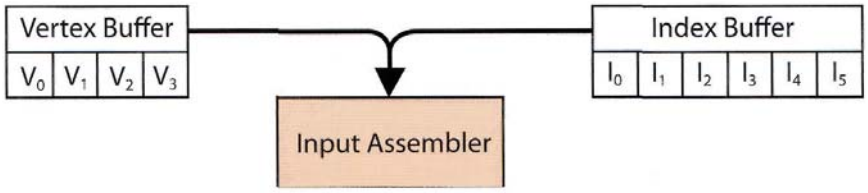

Figure 4.3. Input assembler stage.

4.2.1 Input Assembler

With Direct3D 11 it is now possible to create primitives with up to 32 vertices. But regardless of this, the input assembler functions exactly as it did in previous versions. It uses the vertex declaration (an ID3DllInputLayout created from an array of D3D11_INPUT_ELEMENT_DESCs), a vertex buffer (an ID3DllBuffer with binding of D3D11_BIND_VERTEX_BUFFER) and an index buffer (another ID3DllBuf f er with a binding of D3D11_BIND_INDEX_BUF FER) (see Figure 4.3). The topology set on the device will be D3Dll_PRIMITIVE_T0P0L0GY_n_CONTROL_POINT_PATCHLIST, where n is between 1 and 32. The input assembler will then read the index buffer in chunks of n and will pick out the appropriate vertices from the vertex buffer.

4.2.2 Vertex Shader

Chapter 3 introduced the subtle difference between *control points* and actual vertices; with tessellation, the vertex shader no longer has to output a clip-space vertex to SV_Position, as in Direct3D 10 (see Figure 4.4). (Technically, this could be done in a geometry shader in D3D10, but it was often more efficient to stick with the conventional vertex shader approach.) It is now completely free to operate on data in whatever form the application gives it (through the input assembler) and output that data in whatever coordinate system or format it chooses.

The common use-case for a vertex shader with D3D11 tessellation will be for animation—transforming a model according to the bones provided for a skeletal animation is a good example (and will be covered in more detail as part of Chapter 8). In this example, the vertex shader simply transforms the model-space vertex buffer data into world-space for the later stages. Note that once the vertex shader has

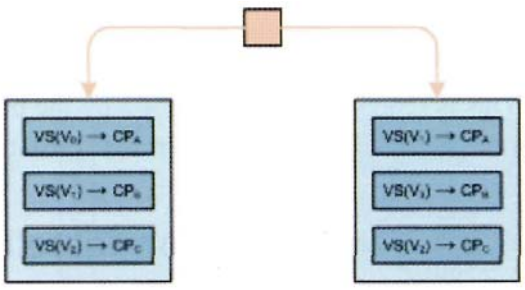

Figure 4.4. Vertex shader stage.

executed, the later stages cannot see any of the original data from the vertex buffer, so if any of this source data is useful, it should be passed down as part of an output.

4.2.3 Hull Shader

This is the first new programmable unit in the Direct3D 11 pipeline. It is made up of two developer-authored functions—the hull shader itself and a "constant function."

The constant function is executed once per patch. Its job is to compute any values that are shared by all control points and that don't logically belong as per-control-point attributes. As far as Direct3D 11 is concerned, the requirement is to output an array of SV_TessFactor and SV_InsideTessFactor values. The size of these arrays varies, depending on the primitive topology defined in the input assembler stage, details of which are discussed later in this chapter. The outputs from a constant function are limited to 128 scalars (32 float-4s), which provides ample room for any additional per-patch constants once the tessellation factors have been included.

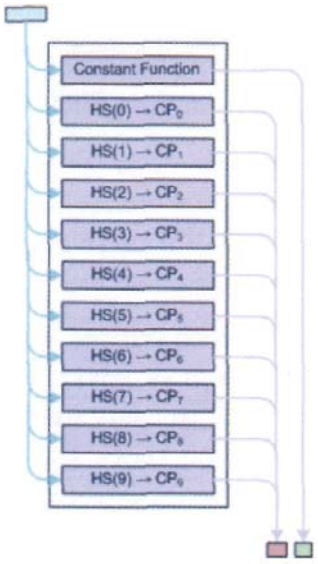

Figure 4.5. Hull shader stage.

An attribute on the main hull shader function declares how many output control points will be generated. This can be a maximum of 32 and does not have to match the topology set on the input assembler—it is perfectly legal for the hull shader to increase or decrease the number of control points. This attribute determines how many times the individual hull shader is executed, once for each of the declared output control points (the index is provided through SV_OutputControlPointID uint input). The quantity of data can be up to 32 float4sor 128 scalars, the same as for the per-patch constant function, but with one difference; the maximum output for all hull shader invocations triggered by a given input patch is 3,968 scalars. In practice, this means that if you're outputting 32 control points, you can only use 31 float4s, instead of 32. Putting these numbers together, the entire hull shader output size is clamped to 4 KB.

Both functions have full visibility of all vertices output by the vertex shader and deemed to be part of this primitive. This is represented in Figure 4.5 by the blue arrows leading from the Vertex Shading section to each of the hull shader and constant function invocations.

4.2.4 Fixed-Function Tessellator

The next stage of processing is entirely fixed function and operates as a black-box, except for the two inputs—the SV_TessFactor and SV_InsideTessFactor values output by the hull shader constant function (see Figure 4.6).

A very important point to note is that the control points output by the hull shader are not used by this stage. That is, it does all of its tessellation work based on the two aforementioned inputs. The control points are there for you as the developer to create your tessellation algorithm (they reappear again as an input to the domain shader), and the fixed-function stage doesn't pay them any attention. This is why there are no restrictions (other than storage) or requirements on the output from the individual invocations.

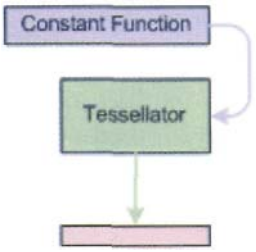

Figure 4.6. Fixed function stage.

The output from the tessellator is a set of weights corresponding to the primitive topology declared in the hull shader—line, triangle or quad. Each of these weights (presented as the SV_DomainLocation input) is fed into a separate domain shader invocation, which is discussed next. In addition to these newly created vertices, which the developer actually sees as part of the domain shading stage, the tessellator also handles the necessary winding and relations between domain samples, so that they form correct triangles that can later be rasterized.

4.2.5 Domain Shader

Domain shader invocations run in isolation, but as is necessary, they can see all the control points and per-patch constants output earlier in the pipeline by the hull shader stage. Simply put, the domain shader's job is to take the point on the line/triangle/quad domain provided by the tessellator and use the control points provided by the hull shader to create a complete new, renderable vertex (see Figure 4.7).

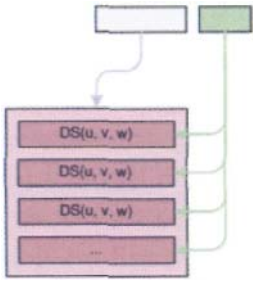

Figure 4.7. Domain shader stage.

It is now the domain shader's responsibility to output a clip-space vertex coordinate to SV_Position. (Strictly speaking, the geometry shader could do this, but it will be less efficient.)

4.2.6 Geometry Shader

With regard to tessellation, this stage remains unchanged from previous versions of Direct3D (see Chapter 3 for details on refinements affecting other aspects of Direct3D rendering) and effectively marks the end of any tessellation related programming. Unless the domain shader has passed along any of the control mesh

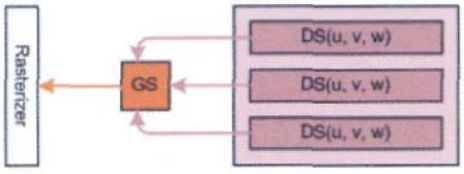

Figure 4.8. Geometry shader stage.

information as per-vertex attributes, the geometry shader has no knowledge of the tessellation work that preceded it.

Unlike in Direct3D 10, where the number of geometry shader invocations was directly linked to the parameters of a draw call by the number of primitives passed into the pipeline, the number of executions when tessellation is enabled is now linked to the SV_TessFactor and SV_InsideTessFactor values emitted from the hull shader constant function (see Figure 4.8). If these are constant and/or set by the application (such as through a constant buffer), you can derive the number of geometry shader invocations, but if a more intelligent LOD scheme is implemented, the number of invocations will be much more difficult to predict.

4.2.7 Rasterization and the Pixel Shaders

Tessellation is a geometry based operation. This means that the final rasterization stages remain completely unchanged and oblivious to anything that came before it.

4.2.8 Demonstration of Possible Outcomes

Earlier in this section, Figure 4.2 showed the "plain" 4-vertex quad of control points. This is the original data sent by the application. The result of the above flow of execution through the Direct3D 11 pipeline transforms Figure 4.9 into Figure 4.10.

In Figure 4.10, you can see the four triangles generated for each input triangle, and you can see that for a 2.0 integer partitioning, it simply splits each edge in half. The next section will detail the tessellation parameters.

Figure 4.11 demonstrates tessellation in a more complex setting. Taken from the DetailTessellation11 sample in the Microsoft DirectX SDK, it shows the result as it would be seen in the final rendered image, as well as the wireframe representation, which

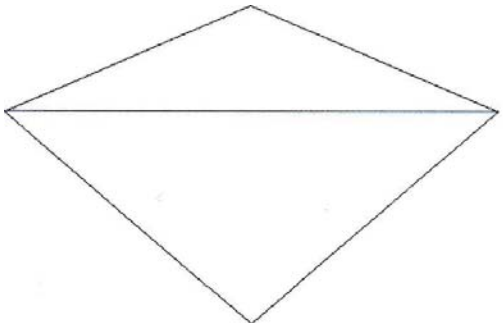

Figure 4.9. Without tessellation enabled.

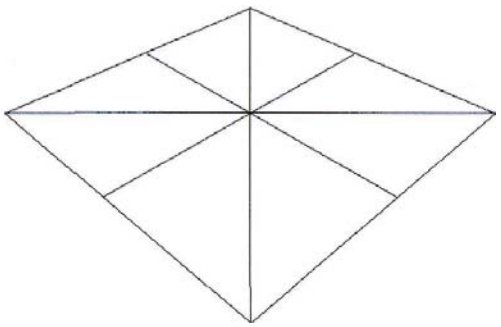

Figure 4.10. With tessellation enabled.

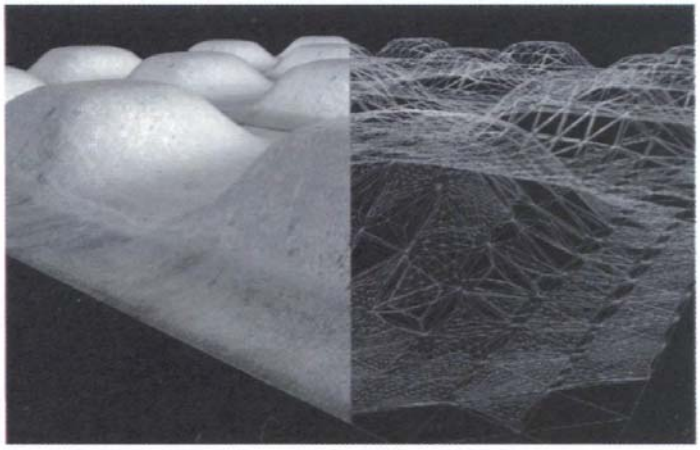

Figure 4.11. DetailTessellation11 from the Microsoft DirectX SDK.

depicts the various tessellation levels (contrast the foreground and background) defined by the underlying geometry.

Chapter 9 covers examples of tessellation algorithms, and the Microsoft DirectX SDK includes several others.

4.3 Parameters for Tessellation

In the preceding section, the detailed flow of execution and data revealed two stages where amplification could occur. First, the hull shader could alter the number of control points passed down the pipeline (3 vertices of a triangle become 10 control points on a cubic surface). Secondly, the fixed-function tessellator generated a number of new primitives, according to the output from the hull shader constant function (7 vertices defining 6 triangles).

As the author of Direct3D 11 shaders, you have full control over these amplification steps.

Generally speaking, there are two scales to consider when deciding the level of amplification—quality and performance. In most cases, these will be inversely proportional, such that higher quality implies greater amplification, which requires more processing and hence reduces performance.

43.1 A Simple Example

Figure 4.12 demonstrates the performance-versus-quality decision. The green curve behind the black arrows can be considered as the ideal surface—the one that an artist or

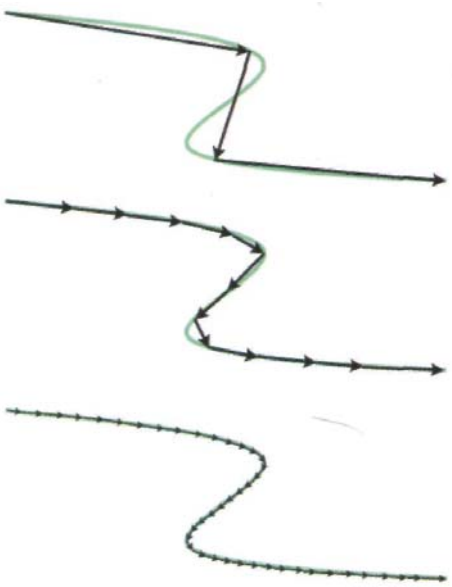

Figure 4.12. Approximations of a smooth curve.

designer would like to have displayed on screen; the black arrows are the line segments generated by the Direct3D 11 pipeline to try to approximate this surface.

The top example generates only 3 line segments. It is quite clear that the end result is not a great approximation. The middle example uses 12 segments and does a much better job of representing the ideal surface; in particular, it only really misses detail in the tight second turn of the surface. The bottom example uses 42 segments and is a near-perfect enough representation of the ideal surface.

Depending on requirements, the above example would ideally use somewhere between 10 and 40 lines to represent the surface. Less than 10, and the approximation would simply not be close enough; above 40 and there will be diminishing returns—extra processing work for little or no visible improvement.

A clever software algorithm would scale between 10 and 40, according to some heuristic of the contribution to the final image. When this particular curve is dominant (as in the center foreground) it could use the full 40 segments, whereas the opposite case (in the far distance off to one side of the image) it would use only 10. An equally clever algorithm may use a performance-based heuristic—if overall performance drops below a certain threshold, it will reduce the quality of the tessellation until performance increases again.

4.3.2 Defining the Tessellation Parameters

The fixed-function tessellation stage uses values from the hull shader's constant function to decide the level of tessellation for a primitive. The following is a simple example of such a function:

```
struct HS_PER_PATCH_OUTPUT
{
    float edgeTesselation[3] : SV_TessFactor;
    float insideTesselation[1] : SV_InsideTessFactor;
};
HS_PER_PATCH_OUTPUT hsPerPatch
```

```
    (
        InputPatch<VS_OUTPUT, 3> ip
        , uint PatchID : SV_PrimitiveID
    )
{
    HS_PER_PATCH_OUTPUT o = (HS_PER_PATCH_OUTPUT)0;

    o.edgeTesselation[0]
        = o.edgeTesselation[1]
        = o.edgeTesselation[2]
        = 2.0f;

    o.insideTesselation[0] = 2.0f;
    return o;
}
```

Listing 4.1. A simple hull shader constant function.

The important detail is that the constant function *MUST* output SV_TessFactor and SV_InsideTessFactor values, which specify how finely to tessellate the current primitive. The tessellator can work on three fundamental types of primitives, and the corresponding output structure varies accordingly. Each of these primitive types, the number of components in their tessellation factors, and a brief sample declaration, are provided in Table 4.1.

Primitive Type	SVTessFactor	SV_InsideTessFactor	Example
Line	2	0	`struct HS_PER_PATCH_OUTPUT { float lenAndThick[2] : SVJessFactor; //... other values };`
Triangle	3	1	`struct HS_PER_PATCH_OUTPUT { float edges[3] : SV_TessFactor; float inside[1] : SVJEnsideTessFactor; // ... other values };`
Quad	4	2	`struct HS_PER_PATCH_OUTPUT { float edges[4] : SV_TessFactor; float inside[2] : SVJEnsideTessFactor; // ... other values};`

Table 4.1. Primitive types, tessellation factors, and examples.

The HLSL compiler will consider it an error if the output structure does not match the declared primitive type.

The number of components in SV_TessFactor matches the number of edges the primitive has, except in the case of a line, where the first value is tessellation along the length of the line, and the second value is the thickness of the tessellation.

SVInsideTessFactor makes more sense when seen visually and is covered later in this section.

The algorithms used to determine the tessellation parameters are subject to the common HLSL and shader programming rules. The code must reside in its own function and can use hard-coded constants, constant buffers, or regular buffers, as well as entirely from first principles without external inputs.

The constant function has visibility of all outputs from the vertex shader, but not the final control points as generated by the other part of the hull shader. The previously mentioned example of adaptive tessellation according to final image contribution could be performed with this information—a rough approximation of the final size within the rasterized image could be made, and the tessellation factors adjusted accordingly.

4.3.3 HLSL Helper Functions

For the most part, the selection of tessellation factors is left to the author of the hull shader's constant function. However Shader model 5.0 defines several utility functions that can be useful when determining intermediary values, as well as handling the pow2 partitioning mode. This latter point is important, as the other three partitioning modes do not require these helper methods and are implemented "behind the scenes" by the compiler; pow2 partitioning must be configured through these intrinsic functions. Table 4.2 provides some information.

The minimum, maximum, or average of the edge factors is then combined with the InsideScale parameter to determine the final internal tessellation factor(s) for the primitive being rendered.

In each of the methods listed above, the final three parameters are marked as outputs that can be retrieved and inspected by HLSL code within the hull shader. With knowledge of the partitioning function, it is easy to write a function of the raw (unrounded) and rounded tessellation factors to compute the current interpolation factor, which could be very important for the later domain shader stage. As explained in the next section, there are important cases where knowledge of neighboring tessellation factors is essential to ensure a continuous surface without gaps.

Function Prototype	Description

`void ProcessQuadTessFactors**(`
`float4 RawEdgeFactors,`
`float InsideScale,`
`float4 RoundedEdgeTessFactors,`
`float2 RoundedInsideTessFactors,`
`float2 UnroundedInsideTessFactors`
`);`
	Applies to a quad where the two internal tessellation factors are independent

`void Process2DQuadTessFactors**(`
`float4 RawEdgeFactors,`
`float2 InsideScale,`
`float4 RoundedEdgeTessFactors,`
`float2 RoundedInsideTessFactors,`
`float2 UnroundedInsideTessFactors`
`);`
	Applies to a quad where both internal factors are the same

`void ProcessTriTessFactors**(`
`float3 RawEdgeFactors,`
`float InsideScale,`
`float3 RoundedEdgeTessFactors,`
`float RoundedInsideTessFactor,`
`float UnroundedInsideTessFactor`
`);`
	Applies to all triangles

Where ** refers to one of:

Min: takes the minimum of the provided edge factors
Max: takes the maximum of the provided edge factors
Avg: takes the average of the provided edge factors

Table 4.2. HLSL Helper Functions.

4.4 Effects of Parameters

Since the tessellation factors are simple scalar values that each apply to different portions of a primitive, it can be difficult to visualize what the factors ultimately influence in the output of the tessellation stages. The following section shows the tessellated output related to changes in the parameters provided by the hull shader constant function. For brevity,

only the triangle and quad primitive types are covered, although line primitives share many of the same properties.

It is also important to note that any factors less than or equal to zero (or *NaN)* cull the primitive, which can be useful in more complex or efficient algorithms—why tessellate a back-facing primitive? Simply cull it in the hull shader! In most cases, it simply sets a minimum value of 1.0 for tessellation factors.

4.4.1 Edge Factors

We begin our discussion of the tessellation factors with *edge factors.* As the name suggests, the three or four values provided for triangle or quad domains correspond directly to the three or four edges that these two primitives have. The fixed function tessellator simply divides up each edge, according to the matching tessellation factor.

Given that a common use for this technology is to generate representations of a higher-order surface from a domain of points, it is natural to assume that the *surface area* (key word: *surface)* is of most importance. However, in a practical implementation it is likely that the edge factors will be the ones demanding attention.

The reasons for this are simply that individual pieces of geometry don't exist in isolation, and that many patches, if not all, will have adjacent patches. These patches join up at the edges; thus; for a continuous surface, you absolutely must tessellate the edges that match up. If you were to have two quads share the same edge but have different tessellation factors, you would likely see gaps in the geometry, as demonstrated in Figure 4.13 by the two orange arrows in gaps along a shared edge where tessellation levels clearly don't match.

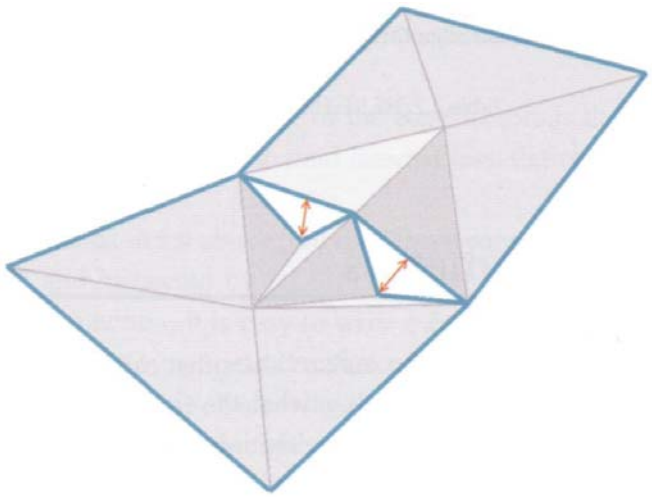

Figure 4.13. Gaps due to mis-matching edge factors.

The term *water tightness* expresses the concern about neighboring patches that share a common edge. Take, for example, a plastic bath toy—a rubber duck—that needs to float in water. If there are any gaps or holes in its surface, it would not be water tight and would sink. Similarly, if this rubber duck were a geometric model being rendered and had gaps between the patches that defined, it we could also declare it as not being water-tight.

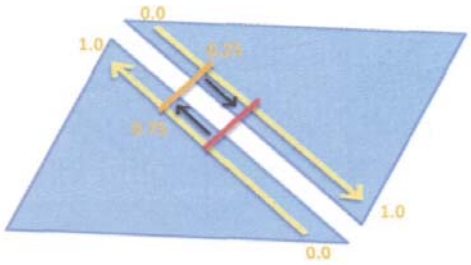

Figure 4.14. Mirrored values on neighboring edge.

There are guarantees from the fixed-function tessellator that assist in this job. First, for a given tessellation factor and partitioning method, it will generate consistent and repeatable distributions of samples—the precision and layout are tightly controlled by both the hardware and software specifications. Second, sample locations are symmetrical about the midpoint of any given edge.

Ultimately, it is responsibility of the author of the domain shader to ensure water tightness. Given *U, UV,* or *UVW*sample locations along a common edge, the domain shader code should generate the same output vertex.

Figure 4.14 demonstrates this mirrored distribution along an edge. Due to vertex ordering, or the geometric layout, it is possible that the *U, UV,* or *UVW* values are reversed in neighboring patches that share a common edge.

Samples are effectively mirrored around the midpoint (0.5 in Figure 4.14) such that a sample at *X* will be the same as the sample at *1.0-X on* an inverted edge.

A simple solution to this problem is to require neighboring patches to use the same tessellation factors along common edges. You then push the problem of blending between different tessellation factors to the inner surface area of the geometry.

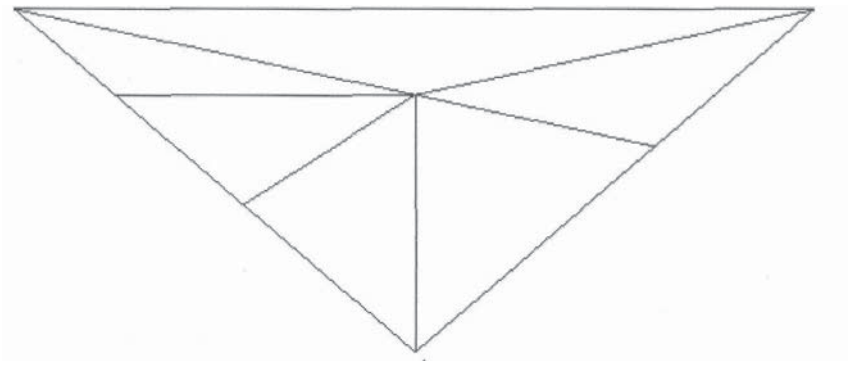

Figure 4.15. Independent edge factors.

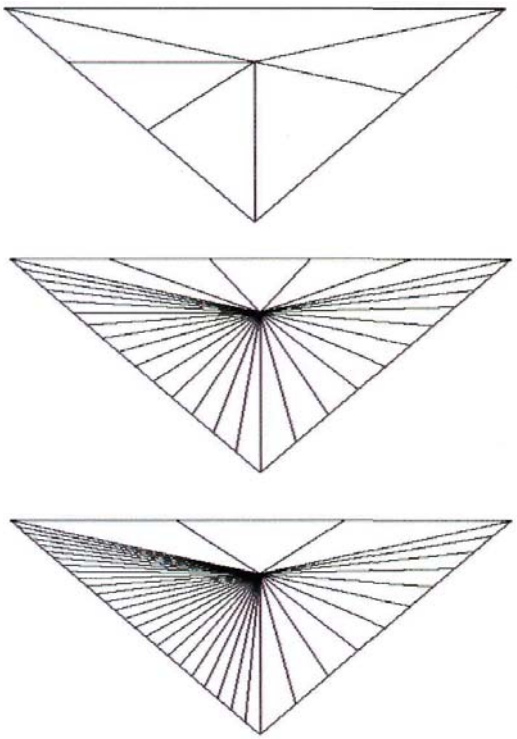

Figure 4.16. Examples of different edge factors.
Top: 1, 2, 3 (clockwise from top)
Middle: 5, 10, 15
Bottom: 3, 10, 25

In more complex settings it is necessary to *blend up* or *blend down* an edge's tessellation factor, based on knowledge of its neighboring patch. Exactly how and when this is done depends entirely on the algorithm being implemented.

Figure 4.15 demonstrates one key detail with edge factors—they are independent. The input mesh is a simple triangle, and the edge factors are 1.0,2.0 and 3.0, clockwise from the top.

Individual edge factors are independent, and there is no technical requirement for water-tight geometry, a restriction that could be prohibitively complex to force upon artists. However, it is a very desirable property to achieve at the algorithmic level, and many implementations pay special attention to ensuring a smooth transition between neighboring primitives.

Figure 4.16 demonstrates several different edge tessellation factors. In all cases, the inside factor is set at 1.0.

4.4.2 Inside Factors

Figure 4.16 depicts varying independent edge factors which reveals one notable shortcoming with edge factors. The relatively small number of vertices generated in the middle of the primitive makes for *a focal point* on the patch. The edges can be highly tessellated, but the surface still always flows through one single central point, giving it a rather significant influence on the final shape and visual appearance of the surface.

The one (triangle) or two (quad) internal tessellation factors can be used help the developer mitigate this problem. Increasing the internal tessellation factors makes the inner area of the surface more densely tessellated.

In all of the cases shown in Figure 4.17, the four edge factors are 1.0, and the blue numbers 1, 3, 6, or 12 indicate the internal tessellation factor. For a quad, both a *U*- and 7-axis of tessellation exist. In the diagram, it can be seen that they operate in essentially the

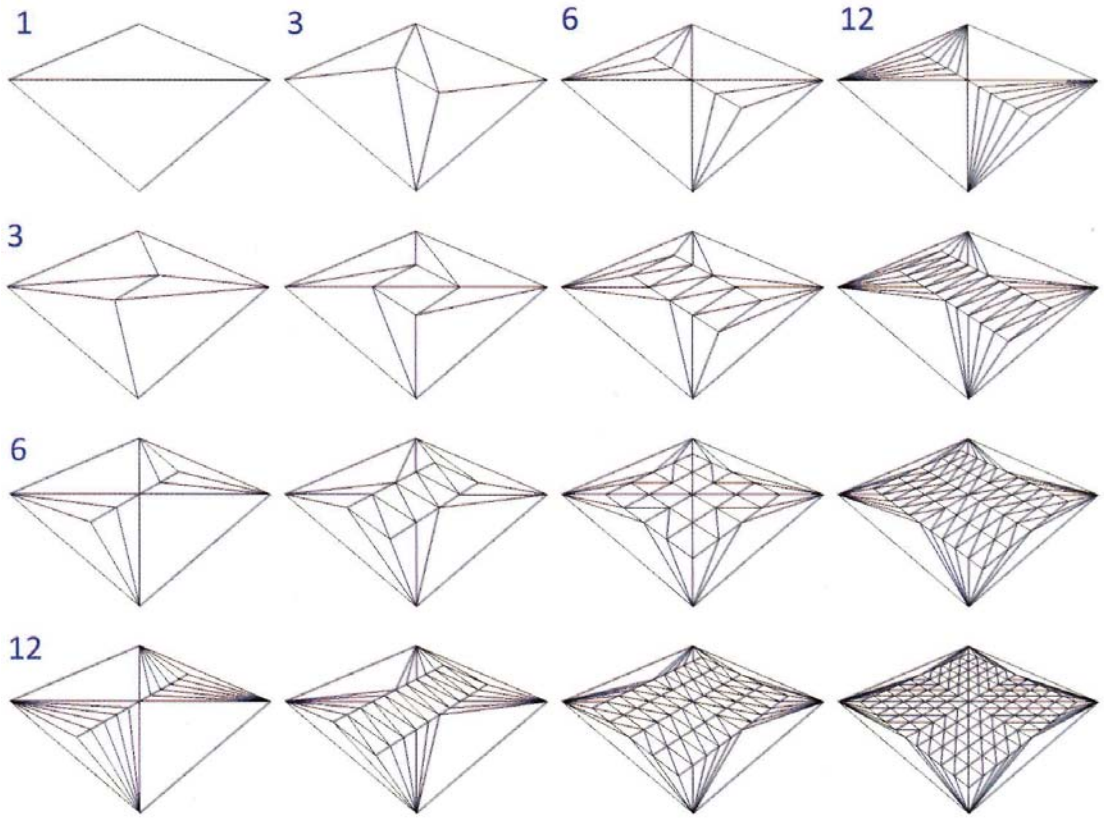

Figure 4.17. Varying internal factors for a quad.

same way, since the bottom-left and top-right halves of the diagram are effectively mirror copies.

Triangles have only one internal tessellation factor, and the surface appears to grow an island of triangles outward from the midpoint as demonstrated in Figure 4.18.

As can be seen in Figure 4.17 and Figure 4.18, having a high internal tessellation factor with a low edge tessellation factor produces the inverse issue that we saw earlier. In this case, the interiors of the primitives are highly tessellated, while they must meet up with very minimal geometry at the edges. Keeping all of the tessellation factors (both internal

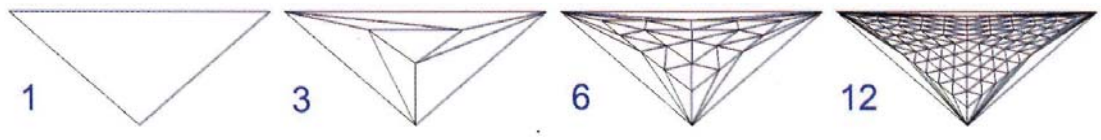

Figure 4.18. Varying internal factors for a triangle.

and external) in a tighter range will greatly improve the final results and avoid the case where factors at opposite extremes come into contact with each other. This will provide a more uniform sampling of the primitive, which in general produces a higher-quality output image.

4.4.3 Partitioning

A detail that has been omitted up to this point is the *partitioning method,* which is declared as an attribute on the main hull shader entry point. This can be integer, fractional_even, fractional_odd, or pow2 and will affect how the fixed-function tessellatOr interprets the tessellation factors provided by the hull shader constant function.

Because this value is declared as a compile-time constant, it will be the same for all patches being rendered with this particular hull shader. This removes any potential hassle with trying to match patch edge factors between different partitioning functions—two quads sharing an edge where each quad defines a different edge factor is tricky enough without the two quads using different internal algorithms.

A key observation is that the factors are specified as *float* values by the constant function, yet it makes no real sense to have a fractional quantity as you either generate a new piece of geometry or don't—you can't create half of a triangle!

Some important considerations include:

- Integer partitioning rounds all floating-point values up to their nearest integer in the range 1 to 64.

  Using f ractional_even rounds to the nearest even number in the range 2 to 64—basically, the series 2, 4, 6, 8, 10 . . . .

- Using f ractional_odd rounds to the nearest odd number in the range 1 to 63—basically, the series 1, 3, 5, 7, 9, 11 ...

  Using pow2 rounds according to the 2n series, where n is an integer between 0 and 7. Basically, the possible post-rounding values are 1, 2, 4, 8, 16, 32, and 64.

Figure 4.19 shows the four partitioning methods available—integer (left), fractional_odd (middle-left), fractional_even (middle-right) and pow2 (right). The vertical axis represents the floating-point tessellation factor provided by the hull shader constant function—the same factor for all six tessellation factors expected for a quad (four edge factors and two internal).

The integer partitioning scheme is straightforward, and the output is as expected—a straight rounding up to the nearest integer. Although there are 9 distinct factors shown (one per row), there are only 3 distinct outputs due to rounding—3, 4, or 5. As soon as the

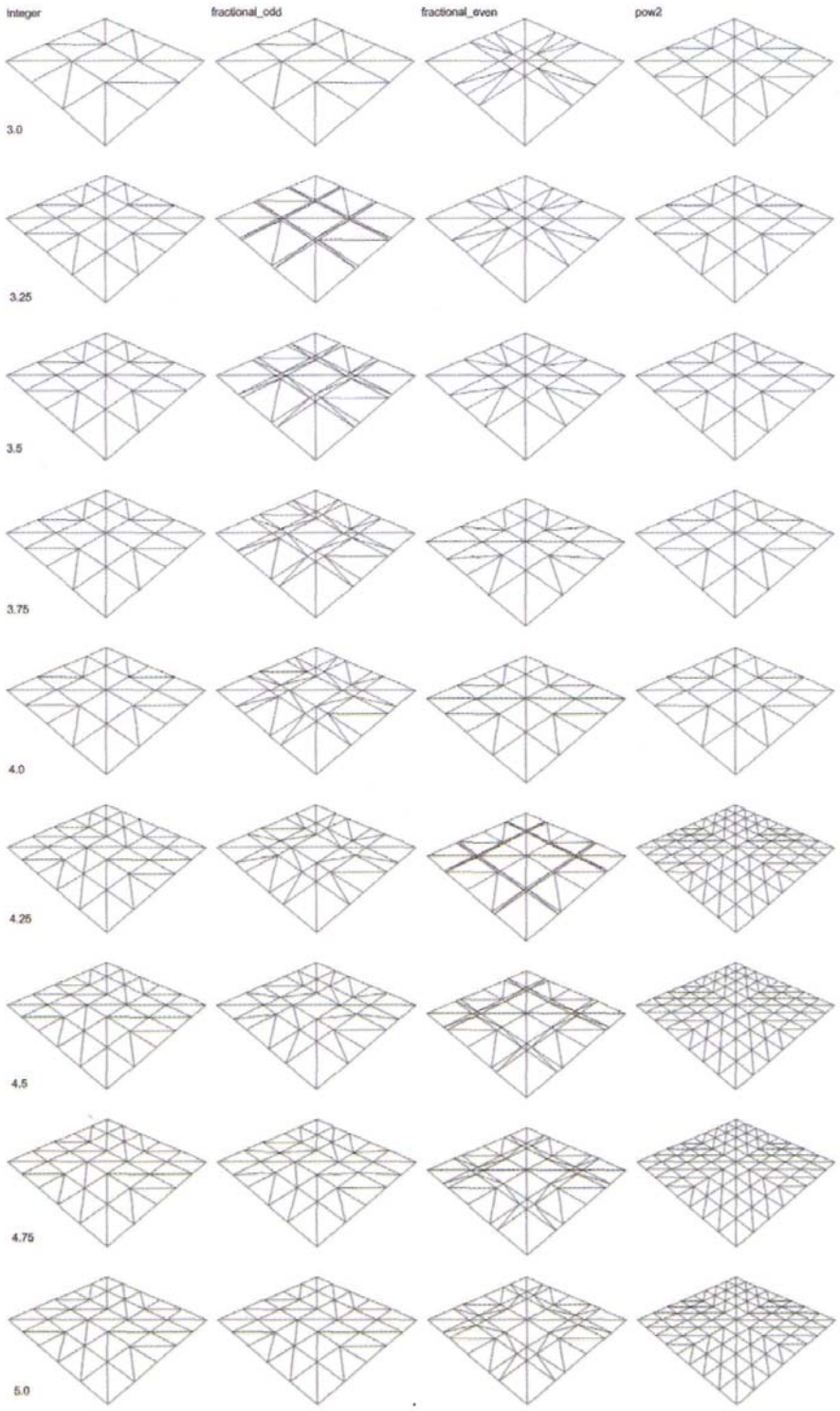

Figure 4.19. Outputs from different partitioning methods.

floating-point value tips over 3.0, the actual value used becomes 4.0. There is no smooth transition, and the change is binary—it's either 3 or 4, nothing else.

The pow2 partitioning is similarly straightforward, in that it rounds completely and immediately up to the next 2" integer value. With 9 distinct tessellation factors, there are only two distinct outputs—4 or 8—and as with integer partitioning, there is no interpolation between these two outputs.

Things get much more interesting with the fractional partitioning functions. A quick glance reveals that the size of some of the rows/columns in the grid of triangles changes with the tessellation factor. A closer look will show that they change smoothly in line with the floating-point factor provided by the hull shader.

As with integer partitioning, the two fractional methods will add new segments as soon as the floating point factor crosses one of the odd or even boundaries, but unlike in the integer mode, it will set two of these segments to be incredibly small and have them slowly grow to the desired size of the next factor up.

Expressing this transition as a percentage makes it easier to understand. For the middle column of Figure 4.19 the "raw" tessellation factors are 3.00, 3.25, 3.50, 3.75, 4.00, 4.25, 4.50, 4.75, and 5.00. But only the first and last images in that sequence have completely finished tessellation (only 3.00 and 5.00 are odd numbers!) so the sequence could be 3.00, 12.5%, 25%, 37.5%, 50.0%, 62.5%, 75.0%, 87.5%, and 4.00—the middle percentages are now more like the weighting used in the common linear interpolation equation. The f ractional_even, right-hand column can be expressed similarly, except that only the 4.00 value in the middle is an even number, and thus of its all segments are equally sized. The right-hand column would become 75%, 87.5%, 4.00, 12.5%, 25%, 37.5%, 50%, and 62.5%.

An interesting observation is that for each integer increment of tessellation factor with f ractional_odd or f ractional_even (such as *2-4* or 3-5) will add two new segments. This makes the morphing pattern much more predictable, since the GPU implementation can place one smaller segment on either side of the reflection/mirror axis. A similar morphing effect with the integer partitioning method would be difficult for the same reasons; adding only one new segment per increment means, for even numbers, that it would have to choose one side of the axis and not be a pure reflection.

Transitioning between Levels of Detail (LODs) for integer partitioning can get tricky, but the relatively large jumps for pow2 partitioning can be very convenient. First, the transition between segments matches the transition between texture mip-map levels (assuming that the convention of 2" texture dimensions is being used), which may be helpful when the domain shader uses a texture to generate the final geometry. Second, each transition doubles the number of segments, which can also be interpreted as splitting each existing segment into two pieces. For custom morphing algorithms this can be a very useful piece of information, since every sample location from the lower/previous level is still present, and all new vertices are equidistant between previous positions, making it relatively simple to interpolate for the new values.

4.4.4 LOD Transitions and the Risk of Popping

Figure 4.19 in the previous section provides a clear view of one potentially big problem regarding the transition between different LODs. For both `integer` and pow2 partitioning, there is a sudden jump where new geometry is added to the final output. In any animated rendering sequence where the LOD is dynamic, this sudden change can be very noticeable to the viewer and generally looks very bad—it is often referred to as *popping*.

Both `fractional_odd` and `fractional_even` have continuous morphing between levels, which greatly reduces this problem. The changing shape of the surface will still be visible, but it will be much less noticeable or irritating. However, it is still not completely immune to displaying the transition between tessellation levels. When the raw input factors are changing in a smooth and continuous manner, the partitioning will handle it well, but if the raw factors jump suddenly, the partitioning will similarly jump, and you'll get a "pop" in the final output.

Temporal considerations are important, here, as this visual artifact is almost always caused by inputs changing through time and across multiple frames in an animation. It is important to think about the selection of tessellation factors with regard to how they change through time, rather than as a single, isolated per-frame factor.