Chapter 8

Primitive Processing

WHAT YOU'LL LEARN IN THIS CHAPTER

- How to use tessellation to add geometric detail to your scenes
- How to use geometry shaders to process whole primitives and create geometry on the fly

In the previous chapters, you've read about the OpenGL pipeline and have been at least briefly introduced to the functions of each of its stages. We've covered the vertex shader stage in some detail, including how its inputs are formed and where its outputs go. A vertex shader runs once on each of the vertices you send OpenGL and produces one set of outputs for each. The next few stages of the pipeline seem similar to vertex shaders at first, but can actually be considered *primitive processing* stages. First, the two tessellation shader stages and the fixed-function tessellator that they flank together process *patches*. Next, the geometry shader processes entire primitives (points, lines, and triangles) and runs once for each. In this chapter, we'll cover both tessellation and geometry shading, and investigate some of the OpenGL features that they unlock.

Tessellation

As introduced in the section "Tessellation" in Chapter 3, tessellation is the process of breaking a large primitive referred to as a *patch* into many smaller primitives before rendering them. There are many uses for tessellation, but the most common application is to add geometric detail to otherwise lower fidelity meshes. In OpenGL, tessellation is produced using three distinct stages of the pipeline — the tessellation control shader (TCS), the fixed-function tessellation engine, and the tessellation evaluation shader (TES). Logically, these three stages fit between the vertex shader and the geometry shader stage. When tessellation is active, incoming vertex data is first processed as normal by the vertex shader and then passed, in groups, to the tessellation control shader.

The tessellation control shader operates on groups of up to 32 vertices[1] at a time, collectively known as a patch. In the context of tessellation, the input vertices are often referred to as *control points*. The tessellation control shader is responsible for generating three things:

- The per-patch inner and outer tessellation factors
- The position and other attributes for each output control point
- Per-patch user-defined varyings

The tessellation factors are sent on to the fixed-function tessellation engine, which uses them to determine the way that it will break up the patch into smaller primitives. Besides the tessellation factors, the output of a tessellation control shader is a new patch (i.e., a new collection of vertices) that is passed to the tessellation evaluation shader after the patch has been tessellated by the tessellation engine. If some of the data is common to all output vertices (such as the color of the patch), then that data may be marked as *per patch*. When the fixed-function tessellator runs, it generates a new set of vertices spaced across the patch as determined by the tessellation factors and the tessellation mode, which is determined using a layout declaration in the tessellation evaluation shader. The only input to the tessellation evaluation shader generated by OpenGL is a set of coordinates indicating where in the patch the vertex lies. When the tessellator is generating triangles, those coordinates are *barycentric*

1. The minimum number of vertices per patch required to be supported by the OpenGL specification is 32. However, the upper limit is not fixed and may be determined by retrieving the value of `GL_MAX_PATCH_VERTICES`.

coordinates. When the tessellation engine is generating lines or triangles, those coordinates are simply a pair of normalized values indicating the relative position of the vertex. This is stored in the `gl_TessCoord` input variable. This setup is shown in the schematic of Figure 8.1.

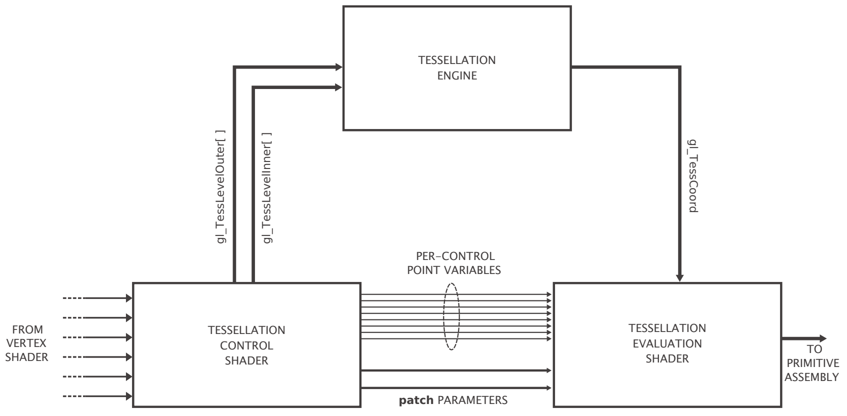

Figure 8.1: Schematic of OpenGL tessellation

Tessellation Primitive Modes

The tessellation mode is used to determine how OpenGL breaks up patches into primitives before passing them on to rasterization. This mode is set using an input layout qualifier in the tessellation evaluation shader and may be one of **`quads`**, **`triangles`**, or **`isolines`**. This primitive mode not only controls the form of the primitives produced by the tessellator, but also the interpretation of the `gl_TessCoord` input variable in the tessellation evaluation shader.

Tessellation Using Quads

When the chosen tessellation mode is set to quads, the tessellation engine will generate a quadrilateral (or quad) and break it up into a set of triangles. The two elements of the `gl_TessLevelInner[]` array should be written by the tessellation control shader and control the level of tessellation applied to the innermost region within the quad. The first element sets the tessellation in the horizontal (u) direction, and the second element sets the tessellation level applied in the vertical (v) direction. Also, all four elements of the `gl_TessLevelOuter[]` array should be written by the tessellation control shader and are used to determine the level of tessellation applied to the outer edges of the quad. This is shown in Figure 8.2.

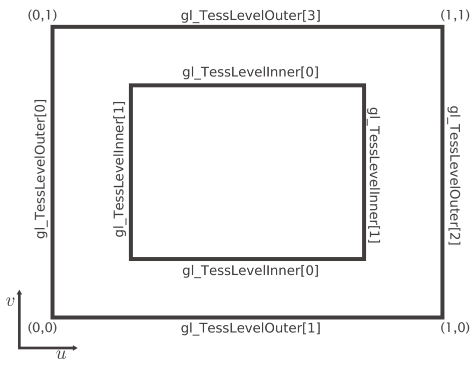

Figure 8.2: Tessellation factors for quad tessellation

When the quad is tessellated, the tessellation engine generates vertices across a two-dimensional domain normalized within the quad. The value stored in the `gl_TessCoord` input variable sent to the tessellation evaluation shader is then a two-dimensional vector (that is, only the x and y components of `gl_TessCoord` are valid) containing the normalized coordinate of the vertex within the quad. The tessellation evaluation shader can use these coordinates to generate its outputs from the inputs passed by the tessellation control shader. An example of quad tessellation produced by the `tessmodes` sample application is shown in Figure 8.3.

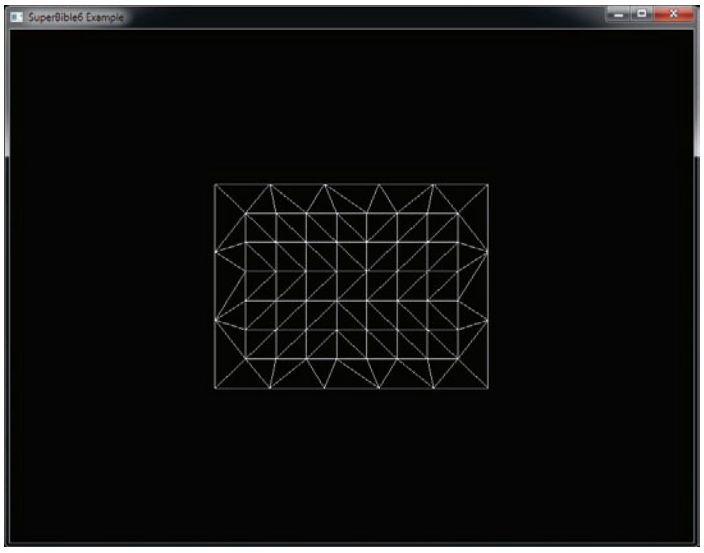

Figure 8.3: Quad tessellation example

In Figure 8.3, the inner tessellation factors in the u and v directions were set to 9.0 and 7.0, respectively. The outer tessellation factors were set to 3.0 and 5.0 in the u and v directions. This was accomplished using the very simple tessellation control shader shown in Listing 8.1.

```
#version 430 core

layout (vertices = 4) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 9.0;
        gl_TessLevelInner[1] = 7.0;
        gl_TessLevelOuter[0] = 3.0;
        gl_TessLevelOuter[1] = 5.0;
        gl_TessLevelOuter[2] = 3.0;
        gl_TessLevelOuter[3] = 5.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.1: Simple quad tessellation control shader example

The result of setting the tessellation factors in this way is visible in Figure 8.3. If you look closely, you will see that along the horizontal outer edges there are five divisions and along the vertical ones there are three divisions. On the interior, you can see that there are 9 divisions along the horizontal axis and 7 along the vertical.

The tessellation evaluation shader that generated Figure 8.3 is shown in Listing 8.2. Notice that the tessellation mode is set using the `quads` input layout qualifier near the front of the tessellation evaluation shader. The shader then uses the x and y components of `gl_TessCoordinate` to perform its own interpolation of the vertex position. In this case, the `gl_in[]` array is four elements long (as specified in the control shader shown in Listing 8.1).

```
#version 430 core

layout (quads) in;

void main(void)
{
    // Interpolate along bottom edge using x component of the
    // tessellation coordinate
    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    // Interpolate along top edge using x component of the
    // tessellation coordinate
    vec4 p2 = mix(gl_in[2].gl_Position,
                    gl_in[3].gl_Position,
                    gl_TessCoord.x);
    // Now interpolate those two results using the y component
    // of tessellation coordinate
    gl_Position = mix(p1, p2, gl_TessCoord.y);
}
```

Listing 8.2: Simple quad tessellation evaluation shader example

Tessellation Using Triangles

When the tessellation mode is set to triangles (again, using an input layout qualifier in the tessellation control shader), the tessellation engine produces a triangle that is then broken into many smaller triangles. Only the first element of the `gl_TessLevelInner[]` array is used, and this level is applied to the entirety of the inner area of the tessellated triangle. The first three elements of the `gl_TessLevelOuter[]` array are used to set the tessellation factors for the three edges of the triangle. This is shown in Figure 8.4.

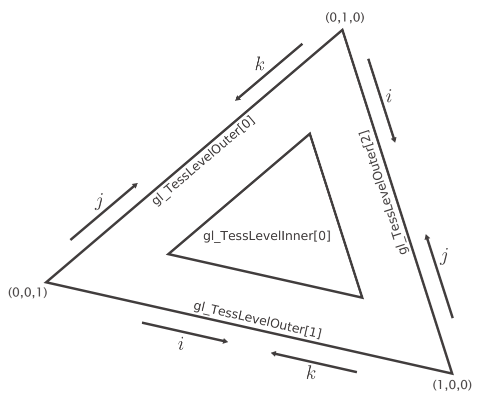

Figure 8.4: Tessellation factors for triangle tessellation

As the tessellation engine generates the vertices corresponding to the tessellated triangles, each vertex is assigned a three-dimensional coordinate called a *barycentric coordinate*. The three components of a barycentric coordinate can be used to form a weighted sum of three inputs representing the corners of a triangle and arrive at a value that is linearly interpolated across that triangle. An example of triangle tessellation is shown in Figure 8.5.

The tessellation control shader used to generate Figure 8.5 is shown in Listing 8.3. Notice how similar it is to Listing 8.1 in that all it does is write

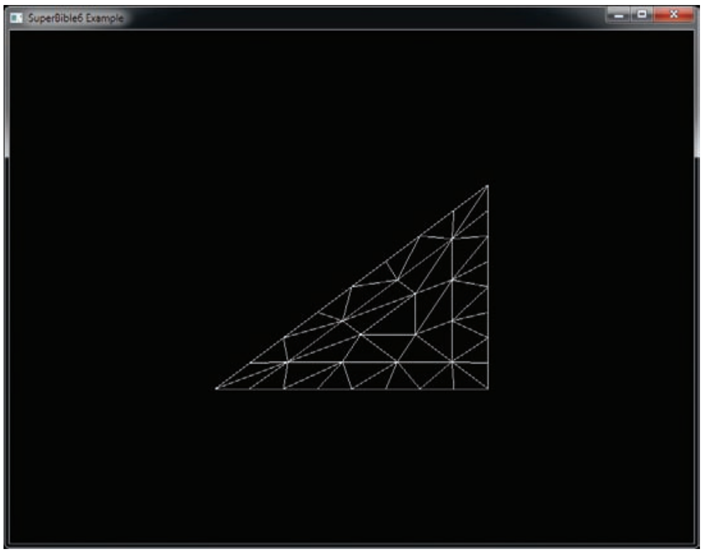

Figure 8.5: Triangle tessellation example

constants into the inner and outer tessellation levels and pass through the control point positions unmodified.

```
#version 430 core

layout (vertices = 3) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 5.0;
        gl_TessLevelOuter[0] = 8.0;
        gl_TessLevelOuter[1] = 8.0;
        gl_TessLevelOuter[2] = 8.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.3: Simple triangle tessellation control shader example

Listing 8.3 sets the inner tessellation level to 5.0 and all three outer tessellation levels to 8.0. Again, looking closely at Figure 8.5, you can see that each of the outer edges of the tessellated triangle has 8 divisions and the inner edges have 5 divisions. The tessellation evaluation shader that produced Figure 8.5 is shown in Listing 8.4.

```
#version 430 core

layout (triangles) in;

void main(void)
{
    gl_Position = (gl_TessCoord.x * gl_in[0].gl_Position) +
                  (gl_TessCoord.y * gl_in[1].gl_Position) +
                  (gl_TessCoord.z * gl_in[2].gl_Position);
}
```

Listing 8.4: Simple triangle tessellation evaluation shader example

Again, to produce a position for each vertex generated by the tessellation engine, we simply calculate a weighted sum of the input vertices. This time, all three components of `gl_TessCoord` are used and represent the relative weights of the three vertices making up the outermost tessellated triangle. Of course, we're free to do anything we wish with the barycentric coordinates, the inputs from the tessellation control shader, and any other data we have access to in the evaluation shader.

Tessellation Using Isolines

Isoline tessellation is a mode of the tessellation engine where, rather than producing triangles, it produces real line primitives running along lines of equal v coordinate in the tessellation domain. Each line is broken up into segments along the u direction. The two outer tessellation factors stored in the first two components of `gl_TessLevelOuter[]` are used to specify the number of lines and the number of segments per line, respectively, and the inner tessellation factors (`gl_TessLevelInner[]`) are not used at all. This is shown in Figure 8.6.

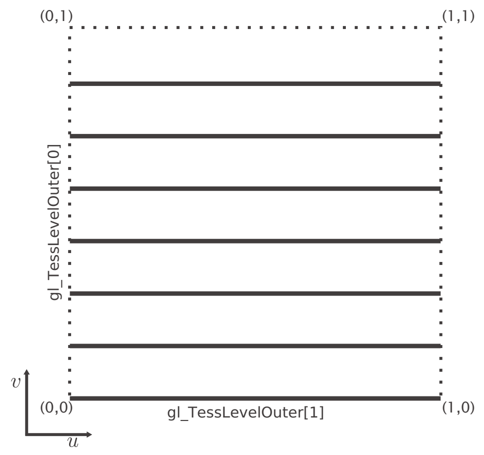

Figure 8.6: Tessellation factors for isoline tessellation

The tessellation control shader shown in Listing 8.5 simply set both the outer tessellation levels to 5.0 and doesn't write to the inner tessellation levels. The corresponding tessellation evaluation shader is shown in Listing 8.6.

```
#version 430 core

layout (vertices = 4) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelOuter[0] = 5.0;
        gl_TessLevelOuter[1] = 5.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.5: Simple isoline tessellation control shader example

Notice that Listing 8.6 is virtually identical to Listing 8.2 except that the input primitive mode is set to `isolines`.

```
#version 430 core

layout (isolines) in;

void main(void)
{
    // Interpolate along bottom edge using x component of the
    // tessellation coordinate
    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    // Interpolate along top edge using x component of the
    // tessellation coordinate
    vec4 p2 = mix(gl_in[2].gl_Position,
                  gl_in[3].gl_Position,
                  gl_TessCoord.x);
    // Now interpolate those two results using the y component
    // of tessellation coordinate
    gl_Position = mix(p1, p2, gl_TessCoord.y);
}
```

Listing 8.6: Simple isoline tessellation evaluation shader example

The result of our extremely simple isoline tessellation example is shown in Figure 8.7.

Figure 8.7 doesn't really seem all that interesting. It's also difficult to see that each of the horizontal lines is actually made up of several segments.

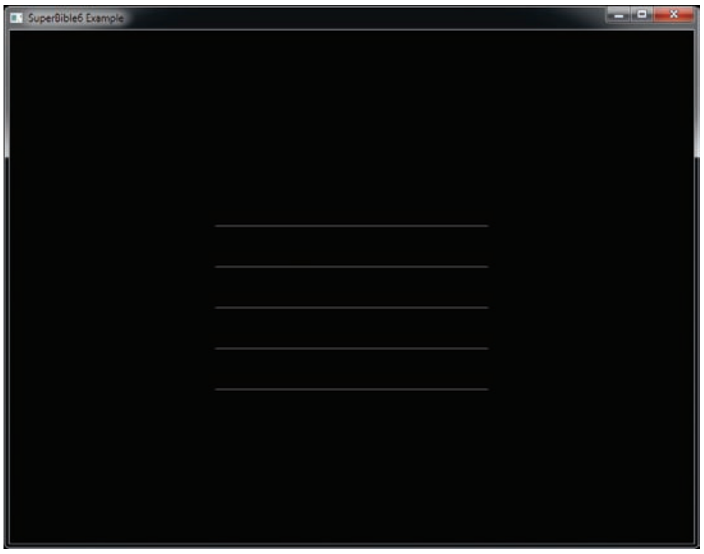

Figure 8.7: Isoline tessellation example

If, however, we change the tessellation evaluation shader to that shown in Listing 8.7, we can generate the image shown in Figure 8.8.

```
#version 430 core

layout (isolines) in;

void main(void)
{
    float r = (gl_TessCoord.y + gl_TessCoord.x / gl_TessLevelOuter[0]);
    float t = gl_TessCoord.x * 2.0 * 3.14159;
    gl_Position = vec4(sin(t) * r, cos(t) * r, 0.5, 1.0);
}
```

Listing 8.7: Isoline spirals tessellation evaluation shader

The shader in Listing 8.7 converts the incoming tessellation coordinates into polar form, with the radius `r` calculated as smoothly extending from zero to one, and with the angle `t` as a scaled version of the x component of the tessellation coordinate to produce a single revolution on each isoline. This produces the spiral pattern shown in Figure 8.8, where the segments of the lines are clearly visible.

Tessellation Point Mode

In addition to being able to render tessellated patches using triangles or lines, it's also possible to render the generated vertices as individual

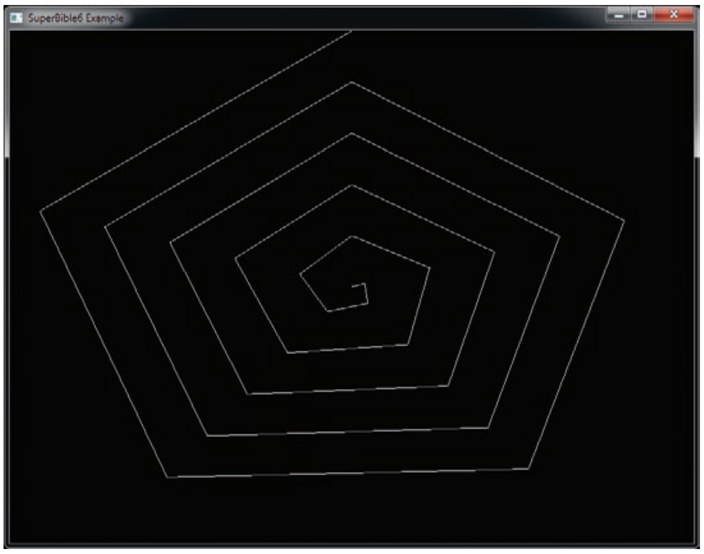

Figure 8.8: Tessellated isoline spirals example

points. This is known as *point mode* and is enabled using the `point_mode` input layout qualifier in the tessellation evaluation shader just like any other tessellation mode. When you specify that point mode should be used, the resulting primitives are points. However, this is somewhat orthogonal to the use of the **`quads`**, **`triangles`**, or **`isolines`** layout qualifiers. That is, you should specify `point_mode` *in addition* to one of the other layout qualifiers. The **`quads`**, **`triangles`**, and **`isolines`** still control the generation of `gl_TessCoord` and the interpretation of the inner and outer tessellation levels. For example, if the tessellation mode is **`quads`**, then `gl_TessCoord` is a two-dimensional vector, whereas if the tessellation mode is **`triangles`**, then it is a three-dimensional barycentric coordinate. Likewise, if the tessellation mode is **`isolines`**, only the outer tessellation levels are used, whereas if it is **`triangles`** or **`quads`**, the inner tessellation levels are used as well.

Figure 8.9 shows a version of Figure 8.5 rendered using point mode next to the original image. To produce the figure on the right, we simply change the input layout qualifier of Listing 8.4 to read:

```
layout (triangles, point_mode) in;
```

As you can see, the layout of the vertices is identical in both sides of Figure 8.9, but on the right, each vertex has been rendered as a single point.

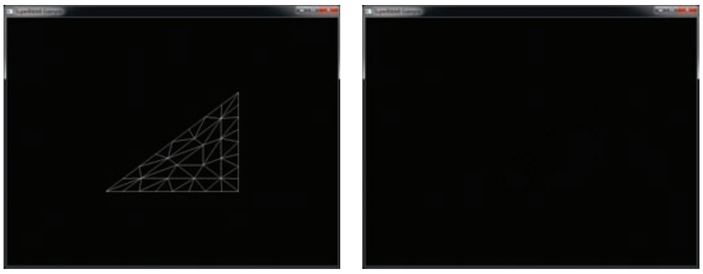

Figure 8.9: Triangle tessellated using point mode

Tessellation Subdivision Modes

The tessellation engine works by generating a triangle or quad primitive and then subdividing its edges into a number of segments determined by the inner and outer tessellation factors produced by the tessellation control shader. It then groups the generated vertices into points, lines, or triangles and sends them on for further processing. In addition to the type of primitives generated by the tessellation engine, you have quite a bit of control about how it subdivides the edges of the generated primitives.

By default, the tessellation engine will subdivide each edge into a number of equal-sized parts where the number of parts is set by the corresponding tessellation factor. This is known as `equal_spacing` mode, and although it is the default, it can be made explicit by including the following layout qualifier in your tessellation evaluation shader:

```
layout (equal_spacing) in;
```

Equal spacing mode is perhaps the easiest mode to comprehend — simply set the tessellation factor to the number segments you wish to subdivide your patch primitive into along each edge, and the tessellation engine takes care of the rest. Although simple, the `equal_spacing` mode comes with a significant disadvantage — as you alter the tessellation factor, it is always rounded up to the next nearest integer and will produce a visible jump from one level to the next as the tessellation factor changes. The two other modes alleviate this problem by allowing the segments to be non-equal in length. These modes are `fractional_even_spacing` and `fractional_odd_spacing`, and again, you can set these modes by using input layout qualifiers as follows:

```
layout (fractional_even_spacing) in;
// or
layout (fractional_odd_spacing) in;
```

With fractional even spacing, the tessellation factor is rounded to the next lower even integer and the edge subdivided as if that were the tessellation factor. With fractional odd spacing, the tessellation factor is rounded down to the next lower odd number and the edge subdivided as if that were the tessellation factor. Of course, with either scheme, there is a small remaining segment that doesn't have the same length as the other segments. That last segment is then cut in half, each half having the same length as the other and is therefore a *fractional* segment.

Figure 8.10 shows the same triangle tessellated with `equal_spacing` mode on the left, `fractional_even_spacing` mode in the center, and `fractional_odd_spacing` mode on the right.

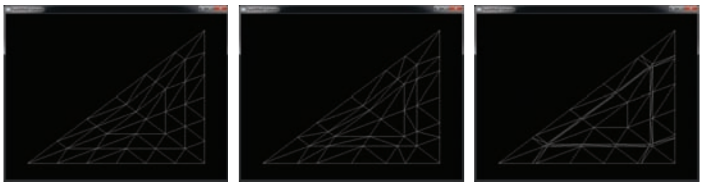

Figure 8.10: Tessellation using different subdivision modes

In all three images shown in Figure 8.10, the inner and outer tessellation factors have been set to 5.3. In the leftmost image showing `equal_spacing` mode, you should be able to see that the number of segments along each of the outer edges of the triangle is 6 — the next integer after 5.3. In the center image, which shows `fractional_even_spacing` spacing, there are 4 equal-sized segments (as 4 is the next lower even integer to 5.3) and then two additional smaller segments. Finally, in the rightmost image, which demonstrates `fractional_odd_spacing`, you can see that there are 5 equal-sized segments (5 being the next lower odd integer to 5.3) and there are two very skinny segments that make up the rest.

If the tessellation level is animated, either by being explicitly turned up and down using a uniform, or calculated in the tessellation control shader, the length of the equal-sized segments and the two filler segments will change smoothly and dynamically. Whether you choose `fractional_even_spacing` or `fractional_odd_spacing` really depends on which looks better in your application — there is generally no real advantage to either. However, unless you need a guarantee that tessellated edges have equal-sized segments and you can live with popping if the tessellation level changes, `fractional_even_spacing` or `fractional_odd_spacing` will generally look better in any dynamic application than `equal_spacing`.

Controlling the Winding Order

In Chapter 3, "Following the Pipeline," we introduced culling and explained how the *winding order* of a primitive affects how OpenGL decides whether to render it. Normally, the winding order of a primitive is determined by the order in which your application presents vertices to OpenGL. However, when tessellation is active, OpenGL generates all the vertices and connectivity information for you. In order to allow you to control the winding order of the resulting primitives, you can specify whether you want the vertices to be generated in clockwise or counterclockwise order. Again, this is specified using an input layout qualifier in the tessellation evaluation shader. To indicate that you want clockwise winding order, use the following layout qualifier:

```
layout (cw) in;
```

To specify that the winding order of the primitives generated by the tessellation engine be counterclockwise, include

```
layout (ccw) in;
```

The `cw` and `ccw` layout qualifiers can be combined with the other input layout qualifiers specified in the tessellation control shader. By default, the winding order is counterclockwise, and so you can omit this layout qualifier if that is what you need. Also, it should be self-evident that winding order only applies to triangles, and so if your application generates isolines or points, then the winding order is ignored — your shader can still include the winding order layout qualifier, but it won't be used.

Passing Data between Tessellation Shaders

In this section, we have looked at how to set the inner and outer tessellation levels for the quad, triangle, and point primitive modes. However, the resulting images in Figures 8.3 through 8.8 aren't particularly exciting, in part because we haven't done anything but compute the positions of the resulting vertices and then just shaded the resulting primitives solid white. In fact, we have rendered all of these images using lines by setting the polygon mode to `GL_LINE` with the **`glPolygonMode()`** function. To produce something a little more interesting, we're going to need to pass more data along the pipeline.

Before a tessellation control shader is run, each vertex represents a control point, and the vertex shader runs once for each input control point and produces its output as normal. The vertices (or control points) are then grouped together and passed together to the tessellation control shader.

The tessellation control shader processes this group of control points and produces a new group of control points that may or may not have the same number of elements in it as the original group. The tessellation control shader actually runs once for each control point in the *output* group, but each invocation of the tessellation control shader has access to all of the input control points. For this reason, both the inputs to and outputs from a tessellation control shader are represented as arrays. The input arrays are sized by the number of control points in each patch, which is set by calling

```
glPatchParameteri(GL_PATCH_VERTICES, n);
```

Here, `n` is the number of vertices per patch. By default, the number of vertices per patch is 3. The size of the input arrays in the tessellation control shader is set by this parameter, and their contents come from the vertex shader. The built-in variable `gl_in[]` is always available and is declared as an array of the `gl_PerVertex` structure. This structure is where the built-in outputs go after you write to them in your vertex shader. All other outputs from the vertex shader become arrays in the tessellation control shader as well. In particular, if you use an output block in your vertex shader, the instance of that block becomes an array of instances in the tessellation control shader. So, for example

```
out VS_OUT
{
    vec4        foo;
    vec3        bar;
    int         baz
} vs_out;
```

becomes

```
in VS_OUT
{
    vec4        foo;
    vec3        bar;
    int         baz;
} tcs_in[];
```

in the tessellation evaluation shader.

The output of the tessellation control shader is also an array, but its size is set by the `vertices` output layout qualifier at the front of the shader. It is quite common to set the input and output vertex count to the same value (as was the case in the samples earlier in this section) and then pass the input directly to the output from the tessellation control shader. However, there's no requirement for this, and the size of the output arrays in the tessellation control shader is limited by the value of the `GL_MAX_PATCH_VERTICES` constant.

As the outputs of the tessellation control shader are arrays, so the inputs to the tessellation evaluation shader are also similarly sized arrays. The tessellation evaluation shader runs once per generated vertex and, like the tessellation control shader, has access to all of the data for all of the vertices in the patch.

In addition to the per-vertex data passed from tessellation control shader to the tessellation evaluation shader in arrays, it's also possible to pass data directly between the stages that is constant across an entire patch. To do this, simply declare the output variable in the tessellation control shader and the corresponding input in the tessellation evaluation shader using the `patch` keyword. In this case the variable does not have to be declared as an array (although you are welcome to use arrays as `patch` qualified variables) as there is only one instance per patch.

Rendering without a Tessellation Control Shader

The purpose of the tessellation control shader is to perform tasks such as computing the value of per-patch inputs to the tessellation evaluation shader and to calculate the values of the inner and outer tessellation levels that will be used by the fixed-function tessellator. However, in some simple applications, there are no per-patch inputs to the tessellation evaluation shader, and the tessellation control shader only writes constants to the tessellation levels. In this case, it's actually possible to set up a program with a tessellation evaluation shader, but without a tessellation control shader.

When no tessellation control shader is present, the default values of all inner and outer tessellation levels is 1.0. You can change this by calling **`glPatchParameterfv()`**, whose prototype is

```
void glPatchParameterfv(GLenum pname,
                        const GLfloat * values);
```

If `pname` is `GL_PATCH_DEFAULT_INNER_LEVEL`, then `values` should point to an array of two floating-point values that will be used as the new default inner tessellation levels in the absence of a tessellation control shader. Likewise, if `pname` is `GL_PATCH_DEFAULT_OUTER_LEVEL`, then `values` should point to an array of four floating-point values that will be used as the new default outer tessellation levels.

If no tessellation control shader is part of the current pipeline, then the number of control points that is presented to the tessellation evaluation shader is the same as the number of control points per patch set by the **`glPatchParameteri()`** when the `pname` parameter is set to

`GL_PATCH_VERTICES`. In this case, the input to the tessellation evaluation shader comes directly from the vertex shader. That is, the input to the tessellation evaluation shader is an array formed from the outputs of the vertex shader invocations that generated the patch.

Communication between Shader Invocations

Although the purpose of output variables in tessellation control shaders is primarily to pass data to the tessellation evaluation shader, they also have a secondary purpose. That is, to communicate data between control shader invocations. As you have read, the tessellation control shader runs a single invocation for each *output* control point in a patch. Each output variable in the tessellation control shader is therefore an array, the length of which is the number of control points in the output patch. Normally, each tessellation control shader invocation will take responsibility for writing to one element of this array.

What might not be obvious is that tessellation control shaders can actually *read* from their output variables — including those that might be written by other invocations! Now, the tessellation control shader is designed in such a way that the invocations can run in parallel. However, there is no ordering guarantee over how those shaders actually execute your code. That means that you have no idea if, when you read from another invocation's output variable, that that invocation has actually written data there.

To deal with this, GLSL includes the `barrier()` function. This is known as a flow-control barrier, as it enforces relative order to the execution of multiple shader invocations. The `barrier()` function really shines when used in compute shaders — we'll get to that later. However, it's available in a limited form in tessellation control shaders, too, with a number of restrictions. In particular, in a tessellation control shader, `barrier()` may only be called directly from within your `main()` function, and can't be inside any control flow structures (such as **`if`**, **`else`**, **`while`**, or **`switch`**).

When you call `barrier()`, the tessellation control shader invocation will stop and wait for all the other invocations in the same patch to catch up. It won't continue execution until all the other invocations have reached the same point. This means that if you write to an output variable in a tessellation control shader and then call `barrier()`, you can be sure that all the other invocations have done the same thing by the time `barrier()` returns, and therefore it's safe to go ahead and read from the other invocations' output variables.

Tessellation Example — Terrain Rendering

To demonstrate a potential use for tessellation, we will cover a simple terrain rendering system based on quadrilateral patches and *displacement mapping*. The code for this example is part of the `dispmap` sample.
A displacement map is a texture that contains the displacement from a surface at each location. Each patch represents a small region of a landscape that is tessellated depending on its likely screen-space area. Each tessellated vertex is moved along the tangent to the surface by the value stored in the displacement map. This adds geometric detail to the surface without needing to explicitly store the positions of each tessellated vertex. Rather, only the displacements from an otherwise flat landscape are stored in the displacement map and are applied at runtime in the tessellation evaluation shader. The displacement map (which is also known as a height map) used in the example is shown in Figure 8.11.

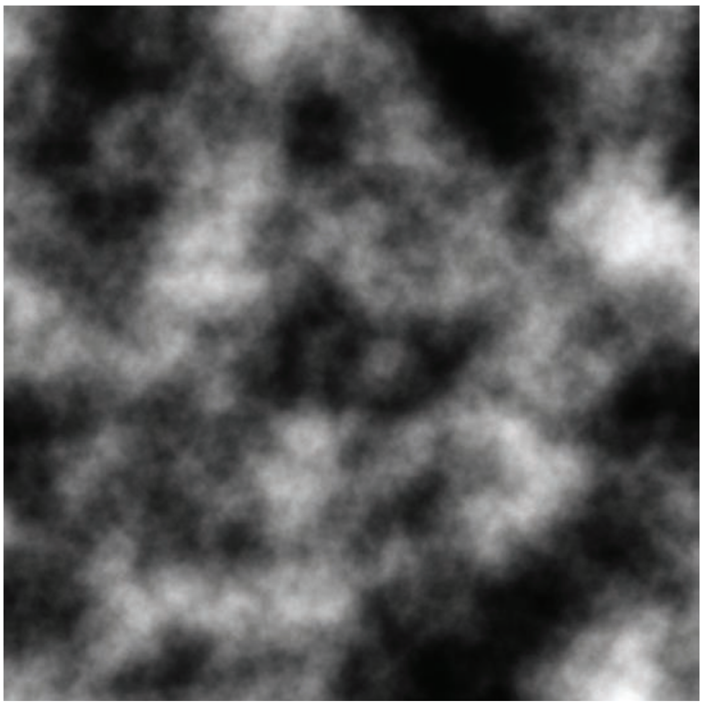

Figure 8.11: Displacement map used in terrain sample

Our first step is to set up a simple vertex shader. As each patch is effectively a simple quad, we can use constants in the shader to represent the four vertices rather than setting up vertex arrays for it. The complete shader is shown in Listing 8.8. The shader uses the instance number (stored in `gl_InstanceID`) to calculate an offset for the patch, which is a one-unit square in the xz plane, centered on the origin. In this application, we will render a grid of 64 × 64 patches, and so the x and y offsets for the patch are calculated by taking `gl_InstanceID` modulo 64 and `gl_InstanceID` divided by 64. The vertex shader also calculates the texture coordinates for the patch, which are passed to the tessellation control shader in `vs_out.tc`.

```
#version 430 core

out VS_OUT
{
    vec2 tc;
} vs_out;

void main(void)
{
    const vec4 vertices[] = vec4[](vec4(-0.5, 0.0, -0.5, 1.0),
                                   vec4( 0.5, 0.0, -0.5, 1.0),
                                   vec4(-0.5, 0.0,  0.5, 1.0),
                                   vec4( 0.5, 0.0,  0.5, 1.0));

    int x = gl_InstanceID & 63;
    int y = gl_InstanceID >> 6;
    vec2 offs = vec2(x, y);

    vs_out.tc = (vertices[gl_VertexID].xz + offs + vec2(0.5)) / 64.0;
    gl_Position = vertices[gl_VertexID] + vec4(float(x - 32), 0.0,
                                               float(y - 32), 0.0);
}
```

Listing 8.8: Vertex shader for terrain rendering

Next, we come to the tessellation control shader. Again, the complete shader is shown in Listing 8.9. In this example, the bulk of the rendering algorithm is implemented in the tessellation control shader, and the majority of the code is only executed by the first invocation. Once we have determined that we are the first invocation by checking that `gl_InvocationID` is zero, we calculate the tessellation levels for the whole patch. First, we project the corners of the patch into normalized device coordinates by multiplying the incoming coordinates by the model-view-projection matrix and then dividing each of the four points by their own homogeneous `.w` component.

Next, we calculate the length of each of the four edges of the patch in normalized device space after projecting them onto the xy plane by ignoring their z components. Then, the shader calculates the tessellation levels of each edge of the patch as a function of its length using a simple scale and bias. Finally, the inner tessellation factors are simply set to the minimum of the outer tessellation factors calculated from the edge lengths in the horizontal or vertical directions.

You may also have noticed a piece of code in Listing 8.9 that checks whether all of the z coordinates of the projected control points are less than zero and then sets the outer tessellation levels to zero if this happens. This is an optimization that culls entire patches that are behind[2] the viewer.

```
#version 430 core

layout (vertices = 4) out;

in VS_OUT
{
    vec2 tc;
} tcs_in[];

out TCS_OUT
{
    vec2 tc;
} tcs_out[];

uniform mat4 mvp;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        vec4 p0 = mvp * gl_in[0].gl_Position;
        vec4 p1 = mvp * gl_in[1].gl_Position;
        vec4 p2 = mvp * gl_in[2].gl_Position;
        vec4 p3 = mvp * gl_in[3].gl_Position;
        p0 /= p0.w;
        p1 /= p1.w;
        p2 /= p2.w;
        p3 /= p3.w;
        if (p0.z <= 0.0 ||
            p1.z <= 0.0 ||
            p2.z <= 0.0 ||
            p3.z <= 0.0)
        {
             gl_TessLevelOuter[0] = 0.0;
             gl_TessLevelOuter[1] = 0.0;
             gl_TessLevelOuter[2] = 0.0;
             gl_TessLevelOuter[3] = 0.0;
        }
        else
        {
            float l0 = length(p2.xy - p0.xy) * 16.0 + 1.0;
            float l1 = length(p3.xy - p2.xy) * 16.0 + 1.0;
            float l2 = length(p3.xy - p1.xy) * 16.0 + 1.0;
            float l3 = length(p1.xy - p0.xy) * 16.0 + 1.0;
            gl_TessLevelOuter[0] = l0;
```

2. This optimization is actually not foolproof. If the viewer were at the bottom of a very steep cliff and looking directly upwards, all four corners of the base patch may be behind the viewer, whereas the cliff cutting through the patch will extend into the viewer's field of view.

```
                gl_TessLevelOuter[1] = l1;
                gl_TessLevelOuter[2] = l2;
                gl_TessLevelOuter[3] = l3;
                gl_TessLevelInner[0] = min(l1, l3);
                gl_TessLevelInner[1] = min(l0, l2);
            }
        }

        gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;
        tcs_out[gl_InvocationID].tc = tcs_in[gl_InvocationID].tc;
    }
```

Listing 8.9: Tessellation control shader for terrain rendering

Once the tessellation control shader has calculated the tessellation levels for the patch, it simply copies its input to its output. It does this per instance and passes the resulting data to the tessellation evaluation shader, which is shown in Listing 8.10.

```
#version 430 core

layout (quads, fractional_odd_spacing) in;

uniform sampler2D tex_displacement;

uniform mat4 mvp;
uniform float dmap_depth;

in TCS_OUT
{
    vec2 tc;
} tes_in[];

out TES_OUT
{
    vec2 tc;
} tes_out;

void main(void)
{
    vec2 tc1 = mix(tes_in[0].tc, tes_in[1].tc, gl_TessCoord.x);
    vec2 tc2 = mix(tes_in[2].tc, tes_in[3].tc, gl_TessCoord.x);
    vec2 tc = mix(tc2, tc1, gl_TessCoord.y);

    vec4 p1 = mix(gl_in[0].gl_Position,
                  gl_in[1].gl_Position,
                  gl_TessCoord.x);
    vec4 p2 = mix(gl_in[2].gl_Position,
                  gl_in[3].gl_Position,
                  gl_TessCoord.x);
    vec4 p = mix(p2, p1, gl_TessCoord.y);

    p.y += texture(tex_displacement, tc).r * dmap_depth;

    gl_Position = mvp * p;
    tes_out.tc = tc;
}
```

Listing 8.10: Tessellation evaluation shader for terrain rendering

The tessellation evaluation shader shown in Listing 8.10 first calculates the texture coordinate of the generated vertex by linearly interpolating the texture coordinates passed from the tessellation control shader of Listing 8.9 (which were in turn generated by the vertex shader of Listing 8.8). It then applies a similar interpolation to the incoming control point positions to produce the position of the outgoing vertex. However, once it's done that, it uses the texture coordinate that it calculated to offset the vertex in the y direction before multiplying that result by the model-view-projection matrix (the same one that was used in the tessellation control shader). It also passes the computed texture coordinate on to the fragment shader in `tes_out.tc`. That fragment shader is shown in Listing 8.11.

```
#version 430 core

out vec4 color;

layout (binding = 1) uniform sampler2D tex_color;

in TES_OUT
{
    vec2 tc;
} fs_in;

void main(void)
{
    color = texture(tex_color, fs_in.tc);
}
```

Listing 8.11: Fragment shader for terrain rendering

The fragment shader shown in Listing 8.11 is really pretty simple. All it does is use the texture coordinate that the tessellation evaluation shader gave it to look up a color for the fragment. The result of rendering with this set of shaders is shown in Figure 8.12.

Of course, if we've done our job correctly, you shouldn't be able to tell that the underlying geometry is tessellated. However, if you look at the wireframe version of the image shown in Figure 8.13, you can clearly see the underlying triangular mesh of the landscape. The goals of the program are that all of the triangles rendered on the screen have roughly similar screen-space area and that sharp transitions in the level of tessellation are not visible in the rendered image.

Tessellation Example — Cubic Bézier Patches

In the displacement mapping example, all we did was use a (very large) texture to drive displacement from a flat surface and then use tessellation

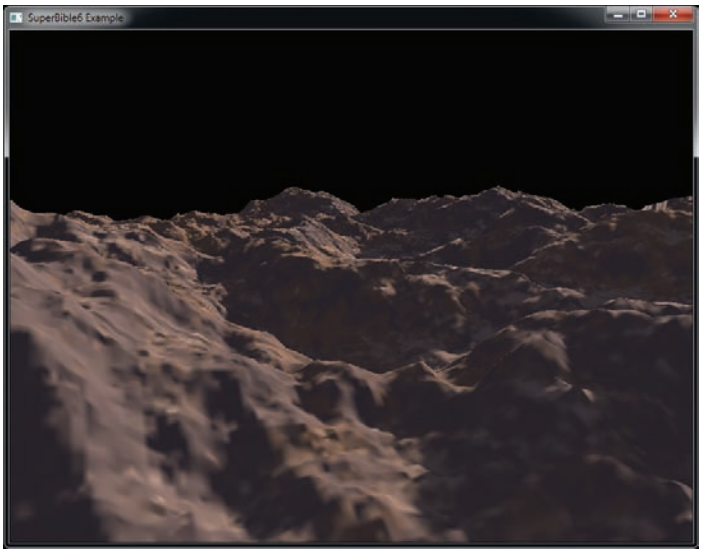

Figure 8.12: Terrain rendered using tessellation

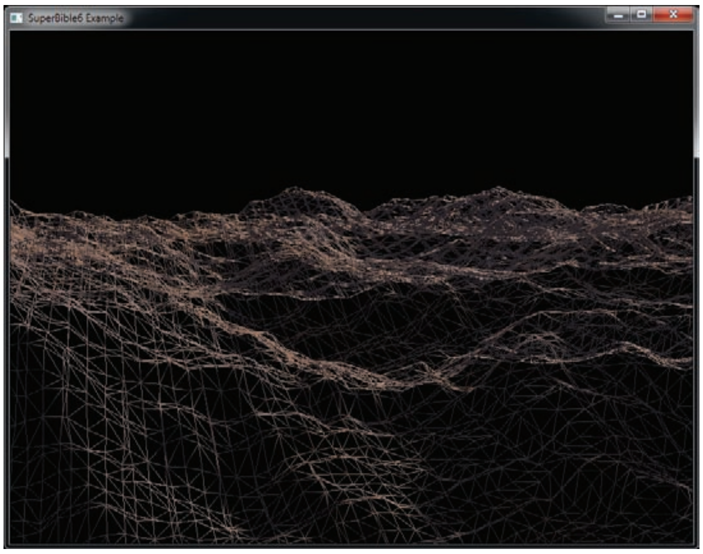

Figure 8.13: Tessellated terrain in wireframe

to increase the number of polygons in the scene. This is a type of brute force, data driven approach to geometric complexity. In the `cubicbezier` example described here, we will use math to drive geometry — we're going to render a *cubic Bézier patch*. If you look back to Chapter 4, you'll see that we've covered all the number crunching we'll need here.

A cubic Bézier patch is a type of *higher order surface* and is defined by a number of *control points*[3] that provide input to a number of interpolation functions that define the surface's shape. A Bézier patch has 16 control points, laid out in a 4×4 grid. Very often (including in this example), they are equally spaced in two dimensions varying only in distance from a shared plane. However, they don't have to be. Free-form Bézier patches are extremely powerful modeling tools, being used natively by many pieces of modeling and design software. With OpenGL tessellation, it's possible to render them directly.

The simplest method of rendering a Bézier patch is to treat the four control points in each row of the patch as the control points for a single cubic Bézier curve, just as was described in Chapter 4. Given our 4×4 grid of control points, we have 4 curves, and if we interpolate along each of them using the same value of t, we will end up with 4 new points. We use these 4 points as the control points for a second cubic Bézier curve. Interpolating along this second curve using a new value for t gives us a second point that lies on the patch. The two values of t (let's call them t_0 and t_1) are the *domain* of the patch and are what is handed to us in the tessellation evaluation shader in `gl_TessCoord.xy`.

In this example, we'll perform tessellation in view space. That means that in our vertex shader, we'll transform our patch's control points into view space by multiplying their coordinates by the model-view matrix — that is all. This simple vertex shader is shown in Listing 8.12.

```
#version 430 core

in vec4 position;

uniform mat4 mv_matrix;

void main(void)
{
    gl_Position = mv_matrix * position;
}
```

Listing 8.12: Cubic Bézier patch vertex shader

3. It should now be evident why the tessellation control shader is so named.

Once our control points are in view space, they are passed to our tessellation control shader. In a more advanced[4] algorithm, we could project the control points into screen space, determine the length of the curve, and set the tessellation factors appropriately. However, in this example, we'll settle with a simple fixed tessellation factor. As in previous examples, we set the tessellation factors only when `gl_InvocationID` is zero, but pass all of the other data through once per invocation. The tessellation control shader is shown in Listing 8.13.

```
#version 430 core

layout (vertices = 16) out;

void main(void)
{
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 16.0;
        gl_TessLevelInner[1] = 16.0;
        gl_TessLevelOuter[0] = 16.0;
        gl_TessLevelOuter[1] = 16.0;
        gl_TessLevelOuter[2] = 16.0;
        gl_TessLevelOuter[3] = 16.0;
    }

    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 8.13: Cubic Bézier patch tessellation control shader

Next, we come to the tessellation evaluation shader. This is where the meat of the algorithm lies. The shader in its entirety is shown in Listing 8.14. You should recognize the `cubic_bezier` and `quadratic_bezier` functions from Chapter 4. The `evaluate_patch` function is responsible for evaluating[5] the vertex's coordinate given the input patch coordinates and the vertex's position within the patch.

```
#version 430 core

layout (quads, equal_spacing, cw) in;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;
```

4. To do this right, we'd need to evaluate the length of the Bézier curve, which involves calculating an integral over a non-closed form... which is hard.

5. You should also now see why the tessellation evaluation shader is so named.

```
out TES_OUT
{
    vec3 N;
} tes_out;

vec4 quadratic_bezier(vec4 A, vec4 B, vec4 C, float t)
{
    vec4 D = mix(A, B, t);
    vec4 E = mix(B, C, t);

    return mix(D, E, t);
}

vec4 cubic_bezier(vec4 A, vec4 B, vec4 C, vec4 D, float t)
{
    vec4 E = mix(A, B, t);
    vec4 F = mix(B, C, t);
    vec4 G = mix(C, D, t);

    return quadratic_bezier(E, F, G, t);
}

vec4 evaluate_patch(vec2 at)
{
    vec4 P[4];
    int i;

    for (i = 0; i < 4; i++)
    {
        P[i] = cubic_bezier(gl_in[i + 0].gl_Position,
                            gl_in[i + 4].gl_Position,
                            gl_in[i + 8].gl_Position,
                            gl_in[i + 12].gl_Position,
                            at.y);
    }

    return cubic_bezier(P[0], P[1], P[2], P[3], at.x);
}

const float epsilon = 0.001;

void main(void)
{
    vec4 p1 = evaluate_patch(gl_TessCoord.xy);
    vec4 p2 = evaluate_patch(gl_TessCoord.xy + vec2(0.0, epsilon));
    vec4 p3 = evaluate_patch(gl_TessCoord.xy + vec2(epsilon, 0.0));

    vec3 v1 = normalize(p2.xyz - p1.xyz);
    vec3 v2 = normalize(p3.xyz - p1.xyz);

    tes_out.N = cross(v1, v2);

    gl_Position = proj_matrix * p1;
}
```

Listing 8.14: Cubic Bézier patch tessellation evaluation shader

In our tessellation evaluation shader, we calculate the surface normal to the patch by evaluating the patch position at two points very close to the point under consideration, using the additional points to calculate two vectors that lie on the patch and then taking their cross product. This is passed to the fragment shader shown in Listing 8.15.

```
#version 430 core

out vec4 color;

in TES_OUT
{
    vec3 N;
} fs_in;

void main(void)
{
    vec3 N = normalize(fs_in.N);

    vec4 c = vec4(1.0, -1.0, 0.0, 0.0) * N.z +
             vec4(0.0, 0.0, 0.0, 1.0);

    color = clamp(c, vec4(0.0), vec4(1.0));
}
```

Listing 8.15: Cubic Bézier patch fragment shader

This fragment shader performs a very simple lighting calculation using the *z* component of the surface normal. The result of rendering with this shader is shown in Figure 8.14.

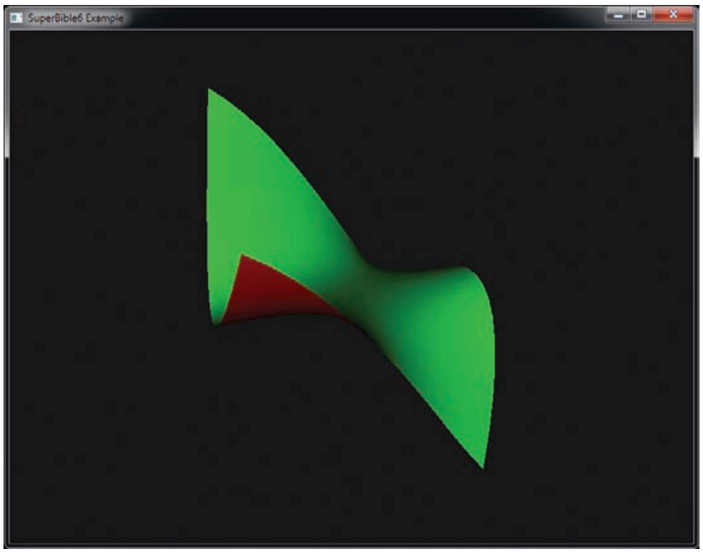

Figure 8.14: Final rendering of a cubic Bézier patch

Because the rendered patch shown in Figure 8.14 is smooth, it is hard to see the tessellation that has been applied to the shape. The left of Figure 8.15 shows a wireframe representation of the tessellated patch, and the right side of Figure 8.15 shows the patch's control points and the control cage, which is formed by creating a grid of lines between the control points.

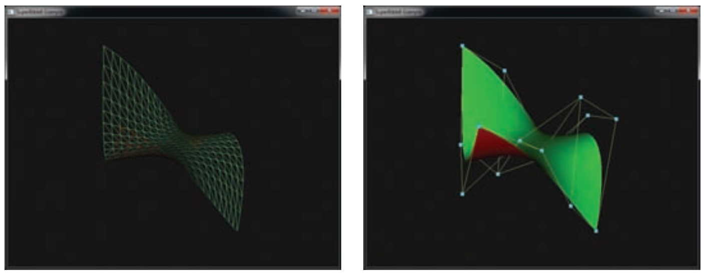
Figure 8.15: A Bézier patch and its control cage

Geometry Shaders

The geometry shader is unique in contrast to the other shader types in that it processes a whole primitive (triangle, line, or point) at once and can actually change the amount of data in the OpenGL pipeline programmatically. A vertex shader processes one vertex at a time; it cannot access any other vertex's information and is strictly one-in, one-out. That is, it cannot generate new vertices, and it cannot stop the vertex from being processed further by OpenGL. The tessellation shaders operate on patches and can set tessellation factors, but have little further control over how patches are tessellated, and cannot produce disjoint primitives. Likewise, the fragment shader processes a single fragment at a time, cannot access any data owned by another fragment, cannot create new fragments, and can only destroy fragments by discarding them. On the other hand, a geometry shader has access to all of the vertices in a primitive (up to six with the primitive modes `GL_TRIANGLES_ADJACENCY` and `GL_TRIANGLE_STRIP_ADJACENCY`), can change the type of a primitive, and can even create and destroy primitives.

Geometry shaders are an optional part of the OpenGL pipeline. When no geometry shader is present, the outputs from the vertex or tessellation evaluation shader are interpolated across the primitive being rendered and are fed directly to the fragment shader. When a geometry shader is present, however, the outputs of the vertex or tessellation evaluation shader become the inputs to the geometry shader, and the outputs of the geometry shader are what are interpolated and fed to the fragment shader. The geometry shader can further process the output of the vertex or tessellation evaluation shader, and if it is generating new primitives (this is called amplification), it can apply different transformations to each primitive as it creates them.

The Pass-Through Geometry Shader

As explained back in Chapter 3, “Following the Pipeline,” the simplest geometry shader that allows you to render anything is the *pass-through* shader, which is shown in Listing 8.16.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;

void main(void)
{
    int i;

    for (i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 8.16: Source code for a simple geometry shader

This is a very simple pass-through geometry shader, which sends its input to its output without modifying it. It looks similar to a vertex shader, but there are a few extra differences to cover. Going over the shader a few lines at a time makes everything clear. The first few lines simply set up the version number (430) of the shader just like in any other shader. The next couple of lines are the first geometry shader-specific parts. They are shown again in Listing 8.17.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip) out;
layout (max_vertices = 3) out;
```

Listing 8.17: Geometry shader layout qualifiers

These set the input and output primitive modes using a layout qualifier. In this particular shader we’re using **`triangles`** for the input and triangle_strip for the output. Other primitive types, along with the layout qualifier, are covered later. For the geometry shader's output, not only do we specify the primitive type, but the maximum number of vertices expected to be generated by the shader (through the max_vertices qualifier). This shader produces individual triangles (generated as very short triangle strips), so we specified 3 here.

Next is our main() function, which is again similar to what might be seen in a vertex or fragment shader. The shader contains a loop, and the loop runs a number of times determined by the length of the built-in array, gl_in. This is another geometry shader-specific variable. Because the geometry shader has access to all of the vertices of the input primitive, the input has to be declared as an array. All of the built-in variables that are written by the vertex shader (such as gl_Position) are placed into a structure, and an array of these structures is presented to the geometry shader in a variable called gl_in.

The length of the gl_in[] array is determined by the input primitive mode, and because in this particular shader, triangles are the input primitive mode, the size of gl_in[] is three. The inner loop is given again in Listing 8.18.

```
for (i = 0; i < gl_in.length(); i++)
{
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
}
```

Listing 8.18: Iterating over the elements of gl_in[]

Inside our loop, we're generating vertices by simply copying the elements of gl_in[] to the geometry shader's output. A geometry shader's outputs are similar to the vertex shader's outputs. Here, we're writing to gl_Position, just as we would in a vertex shader. When we're done setting up all of the new vertex's attributes, we call EmitVertex(). This is a built-in function, specific to geometry shaders that tells the shader that we're done with our work for this vertex and that it should store all that information away and prepare to start setting up the next vertex.

Finally, after the loop has finished executing, there's a call to another special, geometry shader-only function, EndPrimitive(). EndPrimitive() tells the shader that we're done producing vertices for the current primitive and to move on to the next one. We specified triangle_strip as the output for our shader, and so if we continue to call EmitVertex() more than three times, OpenGL continues adding triangles to the triangle strip. If we need our geometry shader to generate separate, individual triangles or multiple, unconnected triangle strips (remember, geometry shaders can create new or amplify geometry), we could call `EndPrimitive()` between each one to mark their boundaries. If you don't call `EndPrimitive()` somewhere in your shader, the primitive is automatically ended when the shader ends.

Using Geometry Shaders in an Application

Geometry shaders, like the other shader types, are created by calling the **`glCreateShader()`** function and using `GL_GEOMETRY_SHADER` as the shader type, as follows:

```
glCreateShader(GL_GEOMETRY_SHADER);
```

Once the shader has been created, it is used like any other shader object. You give OpenGL your shader source code by calling **`glShaderSource()`**, compile the shader using the **`glCompileShader()`** function, and attach it to a program object by calling the **`glAttachShader()`** function. Then the program is linked as normal using the **`glLinkProgram()`** function. Now that you have a program object with a geometry shader linked into it, when you draw geometry using a function like **`glDrawArrays()`**, the vertex shader will run once per vertex, the geometry shader will run once per primitive (point, line, or triangle), and the fragment will run once per fragment. The primitives received by a geometry shader must match what it is expecting based in its own input primitive mode. When tessellation is not active, the primitive mode you use in your drawing commands must match the input primitive mode of the geometry shader. For example, if the geometry shader's input primitive mode is points, then you may only use `GL_POINTS` when you call **`glDrawArrays()`**. If the geometry shader's input primitive mode is triangles, then you may use `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, or `GL_TRIANGLE_FAN` in your **`glDrawArrays()`** call. A complete list of the geometry shader input primitive modes and the allowed geometry types is given in Table 8.1.

Table 8.1: Allowed Draw Modes for Geometry Shader Input Modes

Geometry Shader Input Mode	Allowed Draw Modes
`points`	`GL_POINTS`
`lines`	`GL_LINES`, `GL_LINE_LOOP`, `GL_LINE_STRIP`
`triangles`	`GL_TRIANGLES`, `GL_TRIANGLE_FAN`, `GL_TRIANGLE_STRIP`
`lines_adjacency`	`GL_LINES_ADJACENCY`
`triangles_adjacency`	`GL_TRIANGLES_ADJACENCY`

When tessellation is active, the mode you use in your drawing commands should always be GL_PATCHES, and OpenGL will convert the patches into points, lines, or triangles during the tessellation process. In this case, the input primitive mode of the geometry shader should match the tessellation primitive mode. The input primitive type is specified in the body of the geometry shader using a layout qualifier. The general form of the input layout qualifier is

```
layout (primitive_type) in;
```

This specifies that primitive_type is the input primitive type that the geometry shader is expected to handle, and primitive_type must be one of the supported primitive modes: **points**, **lines**, **triangles**, **lines_adjacency**, or **triangles_adjacency**. The geometry shader runs once per primitive. This means that it'll run once per point for GL_POINTS; once per line for GL_LINES, GL_LINE_STRIP, and GL_LINE_LOOP; and once per triangle for GL_TRIANGLES, GL_TRIANGLE_STRIP, and GL_TRIANGLE_FAN. The inputs to the geometry shader are presented in arrays containing all of the vertices making up the input primitive. The predefined inputs are stored in a built-in array called gl_in[], which is an array of structures defined in Listing 8.19.

```
in gl_PerVertex
{
    vec4  gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
} gl_in[];
```

Listing 8.19: The definition of gl_in[]

The members of this structure are the built-in variables that are written in the vertex shader: gl_Position, gl_PointSize, and gl_ClipDistance[]. You should recognize this structure from its declaration as an output block in the vertex shader described earlier in this chapter. These variables appear as global variables in the vertex shader because the block doesn't have an instance name there, but their values end up in the gl_in[] array of block instances when they appear in the geometry shader. Other variables written by the vertex shader also become arrays in the geometry shader. In the case of individual varyings, outputs in the vertex shader are declared as normal, and the inputs to the geometry shader have a similar declaration, except that they are arrays. Consider a vertex shader that defines outputs as

```
out vec4 color;
out vec3 normal;
```

The corresponding input to the geometry shader would be

```
in vec4 color[];
in vec3 normal[];
```

Notice that both the `color` and `normal` varyings have become arrays in the geometry shader. If you have a large amount of data to pass from the vertex to the geometry shader, it can be convenient to wrap per-vertex information passed from the vertex shader to the geometry shader into an interface block. In this case, your vertex shader will have a definition like this:

```
out VertexData
{
    vec4 color;
    vec3 normal;
} vertex;
```

And the corresponding input to the geometry shader would look like this:

```
in VertexData
{
    vec4 color;
    vec3 normal;
    // More per-vertex attributes can be inserted here
} vertex[];
```

With this declaration, you'll be able to access the per-vertex data in the geometry shader using `vertex[n].color` and so on. The length of the input arrays in the geometry shader depends on the type of primitives that it will process. For example, points are formed from a single vertex, and so the arrays will only contain a single element, whereas triangles are formed from three vertices, and so the arrays will be three elements long. If you're writing a geometry shader that's designed specifically to process a particular primitive type, you can explicitly size your input arrays, which provides a small amount of additional compile-time error checking. Otherwise, you can let your arrays be automatically sized by the input primitive type layout qualifier. A complete mapping of the input primitive modes and the resulting size of the input arrays is shown in Table 8.2.

Table 8.2: Sizes of Input Arrays to Geometry Shaders

Input Primitive Type	Size of Input Arrays
`points`	1
`lines`	2
`triangles`	3
`lines_adjacency`	4
`triangles_adjacency`	6

You also need to specify the primitive type that will be generated by the geometry shader. Again, this is determined using a layout qualifier, like so:

```
layout (primitive_type) out;
```

This is similar to the input primitive type layout qualifier, the only difference being that you are declaring the output of the shader using the out keyword. The allowable output primitive types from the geometry shader are **`points`**, **`line_strip`**, and **`triangle_strip`**. Notice that geometry shaders only support outputting the strip primitive types (not counting points—obviously, there is no such thing as a point strip).

There is one final layout qualifier that must be used to configure the geometry shader. Because a geometry shader is capable of producing a variable amount of data per vertex, OpenGL must be told how much space to allocate for all that data by specifying the maximum number of vertices that the geometry shader is expected to produce. To do this, use the following layout qualifier:

```
layout (max_vertices = n) out;
```

This sets the maximum number of vertices that the geometry shader may produce to n. Because OpenGL may allocate buffer space to store intermediate results for each vertex, this should be the smallest number possible that still allows your application to run correctly. For example, if you are planning to take points and produce one line at a time, then you can safely set this to two. This gives the shader hardware the best opportunity to run fast. If you are going to heavily tessellate the incoming geometry, you might want to set this to a much higher number, although this may cost you some performance. The upper limit on the number of vertices that a geometry shader can produce depends on your OpenGL implementation. It is guaranteed to be at least 256, but the absolute maximum can be found by calling **`glGetIntegerv()`** with the `GL_MAX_GEOMETRY_OUTPUT_VERTICES` parameter.

You can also declare more than one layout qualifier with a single statement by separating them with a comma, like so:

```
layout (triangle_strip, max_vertices = n) out;
```

With these layout qualifiers, a boilerplate **`#version`** declaration, and an empty `main()` function, you should be able to produce a geometry shader that compiles and links but does absolutely nothing. In fact, it will discard any geometry you send it, and nothing will be drawn by your application. We need to introduce two important functions: `EmitVertex()` and `EndPrimitive()`. If you don't call these, nothing will be drawn.

`EmitVertex()` tells the geometry shader that you've finished filling in all of the information for this vertex. Setting up the vertex works much like the vertex shader. You need to write into the built-in variable `gl_Position`. This sets the clip-space coordinates of the vertex that is produced by the geometry shader, just like in a vertex shader. Any other attributes that you want to pass from the geometry shader to the fragment shader can be declared in an interface block or as global variables in the geometry shader. Whenever you call EmitVertex, the geometry shader stores the values currently in all of its output variables and uses them to generate a new vertex. You can call `EmitVertex()` as many times as you like in a geometry shader, until you reach the limit you specified in your **`max_vertices`** layout qualifier. Each time, you put new values into your output variables to generate a new vertex.

An important thing to note about `EmitVertex()` is that it makes the values of any of your output variables (such as `gl_Position`) undefined. So, for example, if you want to emit a triangle with a single color, you need to write that color with every one of your vertices; otherwise, you will end up with undefined results.

`EmitPrimitive()` indicates that you have finished appending vertices to the end of the primitive. Don't forget, geometry shaders only support the strip primitive types (**`line_strip`** and **`triangle_strip`**). If your output primitive type is **`triangle_strip`** and you call `EmitVertex()` more than three times, the geometry shader will produce multiple triangles in a strip. Likewise, if your output primitive type is **`line_strip`** and you call `EmitVertex()` more than twice, you'll get multiple lines. In the geometry shader, `EndPrimitive()` refers to the strip. This means that if you want to draw individual lines or triangles, you have to call `EndPrimitive()` after every two or three vertices. You can also draw multiple strips by calling `EmitVertex()` many times between multiple calls to `EndPrimitive()`.

One final thing to note about calling `EmitVertex()` and `EndPrimitive()` in the geometry shader is that if you haven't produced enough vertices to produce a single primitive (e.g., you're generating **`triangle_strip`** outputs and you call `EndPrimitive()` after two vertices), nothing is produced for that primitive, and the vertices you've already produced are simply thrown away.

Discarding Geometry in the Geometry Shader

The geometry shader in your program runs once per primitive. What you do with that primitive is entirely up to you. The two functions

EmitVertex() and EndPrimitive() allow you to programmatically append new vertices to your triangle or line strip and to start new strips. You can call them as many times as you want (until you reach the maximum defined by your implementation). You're also allowed to not call them at all. This allows you to clip geometry away and discard primitives. If your geometry shader runs and you never call EmitVertex() for that particular primitive, nothing will be drawn. To illustrate this, we can implement a custom backface culling routine that culls geometry as if it were viewed from an arbitrary point in space. This is implemented in the gsculling example.

First, we set up our shader version and declare our geometry shader to accept triangles and to produce triangle strips. Backface culling doesn't really make a lot of sense for lines or points. We also define a uniform that will hold our custom viewpoint in world space. This is shown in Listing 8.20.

```
#version 330

// Input is triangles, output is triangle strip. Because we're going
// to do a 1 in 1 out shader producing a single triangle output for
// each one input, max_vertices can be 3 here.
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;

// Uniform variables that will hold our custom viewpoint and
// model-view matrix
uniform vec3 viewpoint;
uniform mav4 mv_matrix;
```

Listing 8.20: Configuring the custom culling geometry shader

Now inside our main() function, we need to find the face normal for the triangle. This is simply the cross products of any two vectors in the plane of the triangle—we can use the triangle edges for this. Listing 8.21 shows how this is done.

```
// Calculate two vectors in the plane of the input triangle
vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 normal = normalize(cross(ab, ac));
```

Listing 8.21: Finding a face normal in a geometry shader

Now that we have the normal, we can determine whether it faces toward or away from our user-defined viewpoint. To do this, we need to transform the normal into the same coordinate space as the viewpoint, which is world space. Assuming we have the model-view matrix in a uniform, simply multiply the normal by this matrix. To be more accurate, we should multiply the vector by the inverse of the transpose of the upper-left 3×3 submatrix of the model-view matrix. This is known as the normal matrix, and you're free to implement this and put it in its own uniform if you like. However, if your model-view matrix only contains translation, uniform scale (no shear), and rotation, you can use it directly. Don't forget, the normal is a three-element vector, and the model-view matrix is a 4×4 matrix. We need to extend the normal to a four-element vector before we can multiply the two. We can then take the dot product of the resulting vector with the vector from the viewpoint to any point on the triangle.

If the sign of the dot product is negative, that means that the normal is facing away from the viewer and the triangle should be culled. If it is positive, the triangle's normal is pointing toward the viewer, and we should pass the triangle on. The code to transform the face normal, perform the dot product, and test the sign of the result is shown in Listing 8.22.

```
// Calculate the transformed face normal and the view direction vector
vec3 transformed_normal = (vec4(normal, 0.0) * mv_matrix).xyz;
vec3 vt = normalize(gl_in[0].gl_Position.xyz - viewpoint);

// Take the dot product of the normal with the view direction
float d = dot(vt, normal);

// Emit a primitive only if the sign of the dot product is positive
if (d > 0.0)
{
    for (int i = 0; i < 3; i++)
    {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 8.22: Conditionally emitting geometry in a geometry shader

In Listing 8.22, if the dot product is positive, we copy the input vertices to the output of the geometry shader and call `EmitVertex()` for each one. If the dot product is negative, we simply don't do anything at all. This results in the incoming triangle being discarded altogether and nothing being drawn.

In this particular example, we are generating at most one triangle output for each triangle input to the geometry shader. Although the output of the geometry shader is a triangle strip, our strips only contain a single triangle. Therefore, there doesn't strictly need to be a call to `EndPrimitive()`. We just leave it there for completeness.

Figure 8.16 shows a the result of this shader.

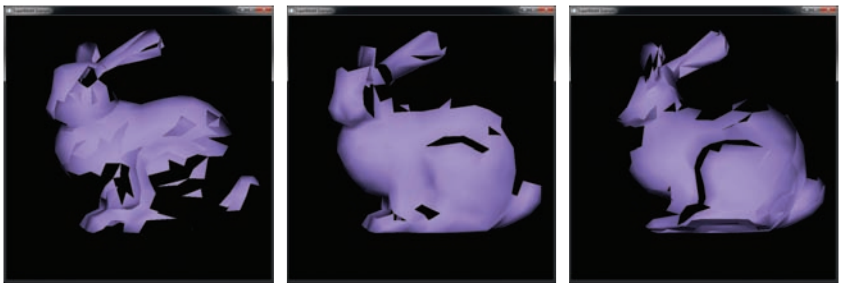

Figure 8.16: Geometry culled from different viewpoints

In Figure 8.16, the virtual viewer has been moved to different positions. As you can see, different parts of the model have been culled away by the geometry shader. It's not expected that this example is particularly useful, but it does demonstrate the ability for a geometry shader to perform geometry culling based on application-defined criteria.

Modifying Geometry in the Geometry Shader

The previous example either discarded geometry or passed it through unmodified. It is also possible to modify vertices as they pass through the geometry shader to create new, derived shapes. Even though your geometry shader is passing vertices on one-to-one (i.e., no amplification or culling is taking place), this still allows you to do things that would otherwise not be possible with a vertex shader alone. If the input geometry is in the form of triangle strips or fans, for example, the resulting geometry will have shared vertices and shared edges. Using the vertex shader to move shared vertices will move all of the triangles that share that vertex. It is not possible, then, to separate two triangles that share an edge in the original geometry using the vertex shader alone. However, this is trivial using the geometry shader.

Consider a geometry shader that accepts triangles and produces **`triangle_strip`** as output. The input to a geometry shader that accepts triangles is individual triangles, regardless of whether they originated from a **glDrawArrays()** or a **glDrawElements()** function call, or whether the primitive type was GL_TRIANGLES, GL_TRIANGLE_STRIP, or GL_TRIANGLE_FAN. Unless the geometry shader outputs more than three vertices, the result is independent, unconnected triangles.

In this next example, we “explode” a model by pushing all of the triangles out along their face normals. It doesn’t matter whether the original model is drawn with individual triangles or with triangle strips or fans. As with the previous example, the input is triangles, the output is **triangle_strip**, and the maximum number of vertices produced by the geometry shader is three because we’re not amplifying or decimating geometry. The setup code for this is shown in Listing 8.23.

```
#version 330

// Input is triangles, output is triangle strip. Because we’re going to do a
// 1 in 1 out shader producing a single triangle output for each one input,
// max_vertices can be 3 here.
layout (triangles) in;
layout (triangle_strip, max_vertices=3) out;
```

Listing 8.23: Setting up the “explode” geometry shader

To project the triangle outward, we need to calculate the face normal of each triangle. Again, to do this we can take the cross product of two vectors in the plane of the triangle—two edges of the triangle. For this, we can reuse the code from Listing 8.21. Now that we have the triangle’s face normal, we can project vertices along that normal by an application-controlled amount. That amount can be stored in a uniform (we call it explode_factor) and updated by the application. This simple code is shown in Listing 8.24.

```
for (int i = 0; i < 3; i++)
{
    gl_Position = gl_in[i].gl_Position +
                  vec4(explode_factor * normal, 0.0);
}
```

Listing 8.24: Pushing a face out along its normal

The result of running this geometry shader on a model is shown in Figure 8.17. The model has been deconstructed, and the individual triangles have become visible.

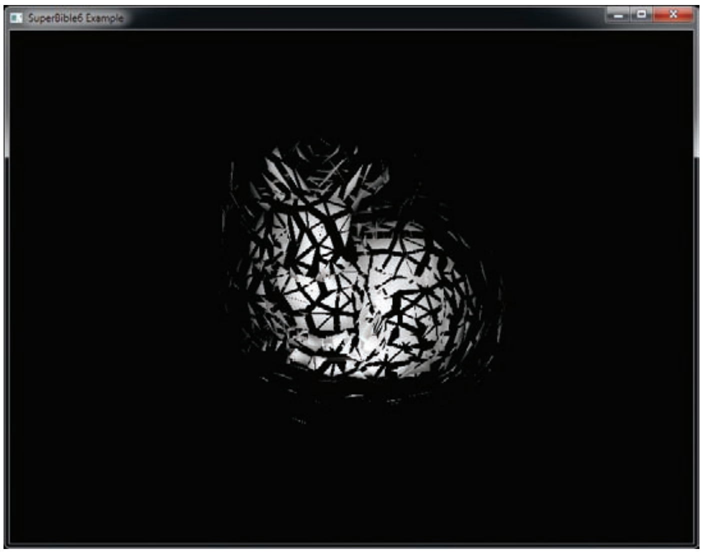

Figure 8.17: Exploding a model using the geometry shader

Generating Geometry in the Geometry Shader

Just as you are not required to call `EmitVertex()` or `EndPrimitive()` at all if you don't want to produce any output from the geometry shader, it is also possible to call `EmitVertex()` and `EndPrimitive()` as many times as you need to produce new geometry. That is, until you reach the maximum number of output vertices that you declared at the start of your geometry shader. This functionality can be used for things like making multiple copies of the input or breaking the input into smaller pieces. This is the subject of the next example, which is the `gstessellate` sample in the book's accompanying source code. The input to our shader is a tetrahedron centered around the origin. Each face of the tetrahedron is made from a single triangle. We tessellate incoming triangles by producing new vertices halfway along each edge and then moving all of the resulting vertices so that they are variable distances from the origin. This transforms our tetrahedron into a spiked shape.

Because the geometry shader operates in object space (remember, the tetrahedron's vertices are centered around the origin), we need to do no coordinate transforms in the vertex shader and, instead, do the transforms in the geometry shader after we've generated the new vertices. To do this, we need a simple, pass-through vertex shader. Listing 8.25 shows a simple pass-through vertex shader.

```
#version 330

in vec4 position;

void main(void)
{
    gl_Position = position;
}
```

Listing 8.25: Pass-through vertex shader

This shader only passes the vertex position to the geometry shader. If you have other attributes associated with the vertices such as texture coordinates or normals, you need to pass them through the vertex shader to the geometry shader as well.

As in the previous example, we accept triangles as input to the geometry shader and produce a triangle strip. We break the strip after every triangle so that we can produce separate, independent triangles. In this example, we produce four output triangles for every input triangle. We need to declare our maximum output vertex count as 12—four triangles times three vertices. We also need to declare a uniform matrix to store the model-view transformation matrix in the geometry shader because we do that transform after generating vertices. Listing 8.26 shows this code.

```
#version 430 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 12) out;

// A uniform to store the model-view-projection matrix
uniform mat4 mvp;
```

Listing 8.26: Setting up the “tessellator” geometry shader

First, let’s copy the incoming vertex coordinates into a local variable. Then, given the original, incoming vertices, we find the midpoint of each edge by taking their average. In this case, however, rather than simply dividing by two, we multiply by a scale factor, which will allow us to alter the *spikiness* of the resulting object. Code to do this is shown in Listing 8.27.

```
// Copy the incoming vertex positions into some local variables
vec3 a = gl_in[0].gl_Position.xyz;
vec3 b = gl_in[1].gl_Position.xyz;
vec3 c = gl_in[2].gl_Position.xyz;
```

```
// Find a scaled version of their midpoints
vec3 d = (a + b) * stretch;
vec3 e = (b + c) * stretch;
vec3 f = (c + a) * stretch;

// Now, scale the original vertices by an inverse of the midpoint
// scale
a *= (2.0 - stretch);
b *= (2.0 - stretch);
c *= (2.0 - stretch);
```

Listing 8.27: Generating new vertices in a geometry shader

Because we are going to generate several triangles using almost identical code, we can put that code into a function (shown in Listing 8.28) and call it from our main tessellation function.

```
void make_face(vec3 a, vec3 b, vec3 c)
{
    vec3 face_normal = normalize(cross(c - a, c - b));
    vec4 face_color = vec4(1.0, 0.2, 0.4, 1.0) * (mat3(mvMatrix) * face_normal
    gl_Position = mvpMatrix * vec4(a, 1.0);
    color = face_color;
    EmitVertex();

    gl_Position = mvpMatrix * vec4(b, 1.0);
    color = face_color;
    EmitVertex();

    gl_Position = mvpMatrix * vec4(c, 1.0);
    color = face_color;
    EmitVertex();

    EndPrimitive();
}
```

Listing 8.28: Emitting a single triangle from a geometry shader

Notice that the `make_face` function calculates a face color based on the face's normal in addition to emitting the positions of its vertices. Now, we simply call `make_face` four times from our main function, which is shown in Listing 8.29.

```
make_face(a, d, f);
make_face(d, b, e);
make_face(e, c, f);
make_face(d, e, f);
```

Listing 8.29: Using a function to produce faces in a geometry shader

Figure 8.18 shows the result of our simple geometry shader-based tessellation program.

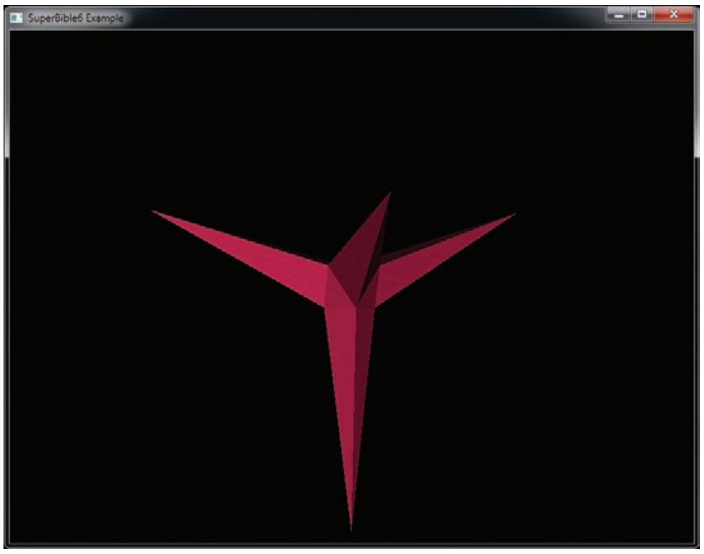

Figure 8.18: Basic tessellation using the geometry shader

Note that using the geometry shader for heavy tessellation may not produce the most optimal performance. If something more complex than that shown in this example is desired, it's best to use the hardware tessellation functions of OpenGL. However, if simple amplification of between two and four output primitives for each input primitive is desired, the geometry shader is probably the way to go.

Changing the Primitive Type in the Geometry Shader

So far, all of the geometry shader examples we've gone through have taken triangles as input and produced triangle strips as output. This doesn't change the geometry type. However, geometry shaders can input and output different types of geometry. For example, you can transform points into triangles or triangles into points. In the `normalviewer` example, which we'll describe next, we're going to change the geometry type from triangles to lines. For each vertex input to the shader, we take the vertex normal and represent it as a line. We also take the face normal and represent that as another line. This allows us to visualize the model's normals—both at each vertex and for each face. Note, though, that if you want to draw the normals on top of the original model, you need to draw everything twice—once with the geometry shader to visualize the normals and once without the geometry shader to show the model. You can't output a mix of two different primitives from a single geometry shader.

For our geometry shader, in addition to the members of the `gl_in` structure, we need the per-vertex normal, and that will have to be passed through the vertex shader. An updated version of the pass-through vertex shader from Listing 8.25 is given in Listing 8.30.

```
#version 330

in vec4 position;
in vec3 normal;

out Vertex
{
    vec3 normal;
}  vertex;

void main(void)
{
    gl_Position = position;
    vertex.normal = normal;
}
```

Listing 8.30: A pass-through vertex shader that includes normals

This passes the position attribute straight through to the `gl_Position` built-in variable and places the normal into an output block.

The setup code for the geometry shader is shown in Listing 8.31. In this example, we accept triangles and produce line strips, each of a single line. Because we output a separate line for each normal we visualize, we produce two vertices for each vertex consumed, plus two more for the face normal. Therefore, the maximum number of vertices that we output per input triangle is eight. To match the Vertex output block that we declared in the vertex shader, we also need to declare a corresponding input interface block in the geometry shader. As we're going to do the object-space-to-world-space transformation in the geometry shader, we declare a mat4 uniform called mvp to represent the model-view-projection matrix. This is necessary so that we can keep the vertex's position in the same coordinate system as its normal until we produce the new vertices representing the line.

```
#version 330

layout (triangles) in;
layout (line_strip) out;
layout (max_vertices = 8) out;

in Vertex
{
    vec3 normal;
}  vertex[];
```

```
// Uniform to hold the model-view-projection matrix
uniform mat4 mvp;

// Uniform to store the length of the visualized normals
uniform float normal_length;
```

Listing 8.31: Setting up the “normal visualizer” geometry shader

Each input vertex is transformed into its final position and emitted from the geometry shader, and then a second vertex is produced by displacing the input vertex along its normal and transforming that into its final position as well. This makes the length of all of our normals one but allows any scaling encoded in our model-view-projection matrix to be applied to them along with the model. We multiply the normals by the application-supplied uniform `normal_length`, allowing them to be scaled to match the model. Our inner loop is shown in Listing 8.32.

```
gl_Position = mvp * gl_in[0].gl_Position;
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();

gl_Position = mvp * (gl_in[0].gl_Position +
                     vec4(gs_in[0].normal * normal_length, 0.0));
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();
EndPrimitive();
```

Listing 8.32: Producing lines from normals in the geometry shader

This generates a short line segment at each vertex pointing in the direction of the normal. Now, we need to produce the face normal. To do this, we need to pick a suitable place from which to draw the normal, and we need to calculate the face normal itself in the geometry shader along which to draw the line.

As in the earlier example given in Listing 8.33, we use a cross product of two of the triangle’s edges to find the face normal. To pick a starting point for the line, we choose the centroid of the triangle, which is simply the average of the coordinates of the input vertices. Listing 8.33 shows the shader code.

```
vec3 ab = gl_in[1].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 ac = gl_in[2].gl_Position.xyz - gl_in[0].gl_Position.xyz;
vec3 face_normal = normalize(cross(ab, ac));

vec4 tri_centroid = (gl_in[0].gl_Position +
                     gl_in[1].gl_Position +
                     gl_in[2].gl_Position) / 3.0;
```

```
gl_Position = mvp * tri_centroid;
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();

gl_Position = mvp * (tri_centroid +
                     vec4(face_normal * normal_length, 0.0));
gs_out.normal = gs_in[0].normal;
gs_out.color = gs_in[0].color;
EmitVertex();
EndPrimitive();
```

Listing 8.33: Drawing a face normal in the geometry shader

Now when we render a model, we get the image shown in Figure 8.19.

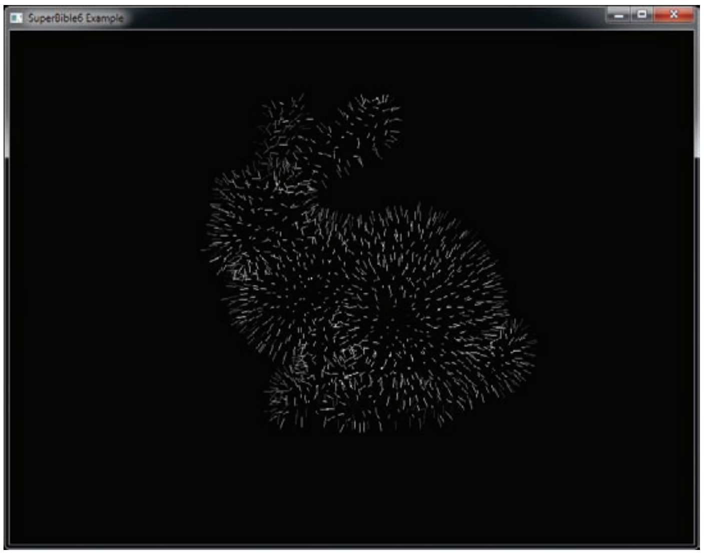

Figure 8.19: Displaying the normals of a model using a geometry shader

Multiple Streams of Storage

When only a vertex shader is present, there is a simple one-in, one-out relationship between the vertices coming into the shader and the vertices stored in the transform feedback buffer. When a geometry shader is present, each shader invocation may store zero, one, or more vertices into the bound transform feedback buffers. Not only this, but it's actually possible to configure up to four output *streams* and use the geometry shader to send its output to whichever one it chooses. This can be used, for example, to sort geometry or to render some primitives while storing other geometry in transform feedback buffers. There are a couple of pretty major limitations when multiple output streams are used in a geometry shader; first, the output primitive mode from the geometry shader for all streams must be set to **points**. Second, although it's possible to simultaneously render geometry and to store data into transform feedback buffers, only the first stream may be rendered — the others are for storage only. If your application fits with these constraints, then this can be a very powerful feature.

To set up multiple output streams from your geometry shader, use the `stream` layout qualifier to select one of four streams. As with most other output layout qualifiers, the `stream` qualifier may be applied directly to a single output or to an output block. It can also be applied directly to the **out** keyword without declaring an output variable, in which case it will affect all further output declarations until another `stream` layout qualifier is encountered. For example, consider the following output declarations in a geometry shader:

```
out vec4                    foo; // "foo" is in stream 0 (the default).
layout (stream=2) out vec4  bar; // "bar" is part of stream 2.
out vec4                    baz; // "baz" is back in stream 0.
layout (stream=1) out;           // Everything from here on is in stream 1.
out int                     apple;  // "apple" and "orange" are part
out int                     orange; // of stream 1.
layout (stream=3) out MY_BLOCK   // Everything in "MY_BLOCK" is in
stream 3.
{
    vec3                    purple;
    vec3                    green;
};
```

In the geometry shader, when you call `EmitVertex()`, the vertex will be recorded into the first output stream (stream 0). Likewise, when you call `EndPrimitive()`, it will end the primitive being recorded to stream 0. However, you can call `EmitStreamVertex()` and `EndStreamPrimitive()`, both of which take an integer argument specifying the stream to send the output to:

```
void EmitStreamVertex(int stream);

void EndStreamPrimitive(int stream);
```

The `stream` argument must be a compile time constant. If rasterization is enabled, then any primitives sent to stream 0 will be rasterized.

New Primitive Types Introduced by the Geometry Shader

Four new primitive types were introduced with geometry shaders: `GL_LINES_ADJACENCY`, `GL_LINE_STRIP_ADJACENCY`, GL_TRIANGLES_ADJACENCY, and GL_TRIANGLE_STRIP_ADJACENCY. These primitive types are really only useful when rendering with a geometry shader active. When the new adjacency primitive types are used, for each line or triangle passed into the geometry shader, it not only has access to the vertices defining that primitive, but it also has access to the vertices of the primitive that is next to the one it's processing.

When you render using GL_LINES_ADJACENCY, each line segment consumes four vertices from the enabled attribute arrays. The two center vertices make up the line; the first and last vertices are considered the adjacent vertices. The inputs to the geometry shader are therefore four-element arrays. In fact, because the input and output types of the geometry shader do not have to be related, GL_LINES_ADJACENCY can be seen as a way of sending generalized four-vertex primitives to the geometry shader. The geometry shader is free to transform them into whatever it pleases. For example, your geometry shader could convert each set of four vertices into a triangle strip made up of two triangles. This allows you to render quads using the GL_LINES_ADJACENCY primitive. It should be noted, though, that if you draw using GL_LINES_ADJACENCY when no geometry shader is active, regular lines will be drawn using the two innermost vertices of each set of four vertices. The two outermost vertices will be discarded, and the vertex shader will not run on them at all.

Using GL_LINE_STRIP_ADJACENCY produces a similar effect. The difference is that the entire strip is considered to be a primitive, with one additional vertex on each end. If you send eight vertices to OpenGL using GL_LINES_ADJACENCY, the geometry shader will run twice, whereas if you send the same vertices using GL_LINE_STRIP_ADJACENCY, the geometry shader will run five times. Figure 8.20 should make things clear. The eight vertices in the top row are sent to OpenGL with the GL_LINES_ADJACENCY primitive mode. The geometry shader runs twice on four vertices each time—ABCD and EFGH. In the second row, the same eight vertices are sent to OpenGL using the GL_LINE_STRIP_ADJACENCY primitive mode. This time, the geometry shader runs five times—ABCD, BCDE, and so on until EFGH. In each case, the solid arrows are the lines that would be rendered if no geometry shader were present.

The GL_TRIANGLES_ADJACENCY primitive mode works similarly to the GL_LINES_ADJACENCY mode. A triangle is sent to the geometry shader for each set of six vertices in the enabled attribute arrays. The first, third, and fifth vertices are considered to make up the real triangle, and the second, fourth, and sixth vertices are considered to be in between the triangle's

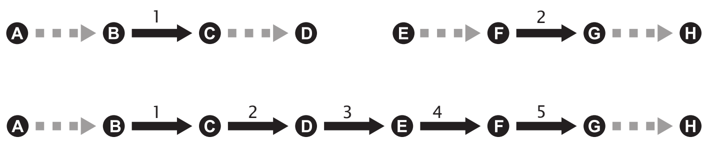

Figure 8.20: Lines produced using lines with adjacency primitives

vertices. This means that the inputs to the geometry shader are six-element arrays. As before, you can do anything you want to the vertices using the geometry shader; `GL_TRIANGLES_ADJACENCY` is a good way to get arbitrary six-vertex primitives into the geometry shader. Figure 8.21 shows this.

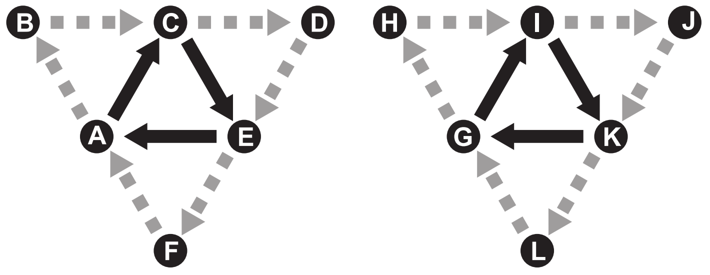

Figure 8.21: Triangles produced using `GL_TRIANGLES_ADJACENCY`

The final, and perhaps most complex (or alternatively the most difficult to understand), of these primitive types is `GL_TRIANGLE_STRIP_ADJACENCY`. This primitive represents a triangle strip with every other vertex (the first, third, fifth, seventh, ninth, and so on) forming the strip. The vertices in between are the adjacent vertices. Figure 8.22 demonstrates the principle. In the figure, the vertices A through P represent 16 vertices sent to OpenGL. A triangle strip is generated from every other vertex (A, C, E, G, I, and so on), and the vertices that come between them (B, D, F, H, J, and so on) are the adjacent vertices.

There are special cases for the triangles that come at the start and end of the strip, but once the strip is started, the vertices fall into a regular pattern that is more clearly seen in Figure 8.23.

The rules for the ordering of `GL_TRIANGLE_STRIP_ADJACENCY` are spelled out clearly in the OpenGL Specification—in particular, the special cases are noted there. You are encouraged to read that section of the specification if you want to work with this primitive type.

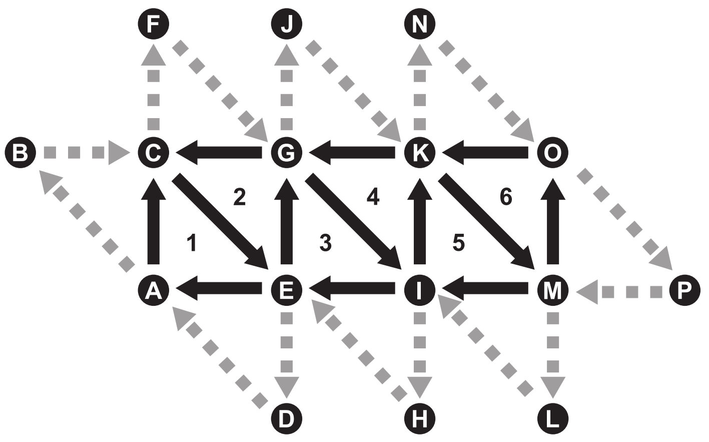

Figure 8.22: Triangles produced using `GL_TRIANGLE_STRIP_ADJACENCY`

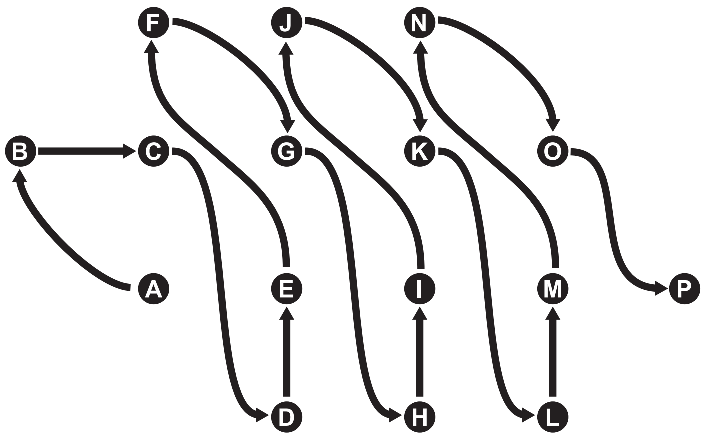

Figure 8.23: Ordering of vertices for `GL_TRIANGLE_STRIP_ADJACENCY`

Rendering Quads Using a Geometry Shader

In computer graphics, the word *quad* is used to describe a quadrilateral – a shape with four sides. Modern graphics APIs do not support rendering quads directly, primarily because modern graphics hardware does not support quads. When a modeling program produces an object made from quads, it will often include the option to export the geometry data by converting each quad into a pair of triangles. These are then rendered by the graphics hardware directly. In some graphics hardware, quads are supported, but internally the hardware will do this conversion from quads to pairs of triangles for you.

In many cases, breaking a quad into a pair of triangles works out just fine and the visual image isn't much different than what would have been rendered had native support for quads been present. However, there are a large class of cases where breaking a quad into a pair of triangles *doesn't* produce the correct result. Take a look at Figure 8.24.

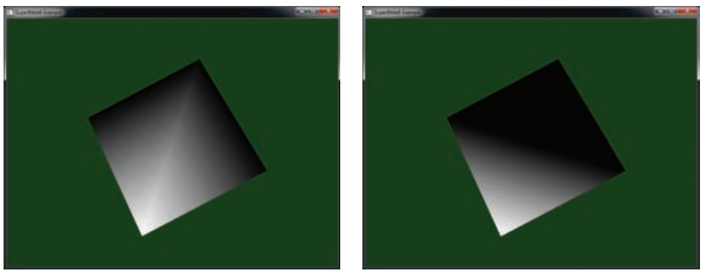

Figure 8.24: Rendering a quad using a pair of triangles

In Figure 8.24, we have rendered a quad as a pair of triangles. In both images, the vertices are wound in the same order. There are three black vertices and one white vertex. In the left image, the split between the triangles runs vertically through the quad. The topmost and two side vertices are black and the bottommost vertex is white. The seam between the two triangles is clearly visible as a bright line. In the right image, the quad has been split horizontally. This has produced the topmost triangle, which contains only black vertices and is therefore entirely black, and the bottommost triangle, which contains one white vertex and two black ones, therefore displaying a black to white gradient.

The reason for this is that during rasterization and interpolation of the per-vertex colors presented to the fragment shader, we're only rendering a triangle. There are only three vertices' worth of information available to us at any given time, and therefore, we can't take into consideration the "other" vertex in the quad.

Clearly, neither image is correct, but neither is obviously better than the other. Also, the two images are radically different. If we rely on our export tools, or worse a runtime library, to split quads for us, we do not have any control over which of these two images we'll get. What can we do about that? Well, the geometry shader is able to accept primitives with the `GL_LINES_ADJACENCY` type, and each of these has four vertices — exactly enough to represent a quad. This means that by using lines with adjacency, we can get four vertices' worth of information at least as far as the geometry shader.

Next, we need to deal with the rasterizer. Recall, the output of the geometry shader can only be points, lines, or triangles, and so the best we can do is to break each quad (represented by a `lines_adjacency` primitive) into a pair of triangles. You might think this leaves us in the same spot as we were before. However, we now have the advantage that we can pass whatever information we like on to the fragment shader.

To correctly render a quad, we must consider the parameterization of the domain over which we want to interpolate our colors (or any other attribute). For triangles, we use barycentric coordinates, which are three-dimensional coordinates used to weight the three corners of the triangle. However, for a quad, we can use a two-dimensional parameterization. Consider the quad shown in Figure 8.25.

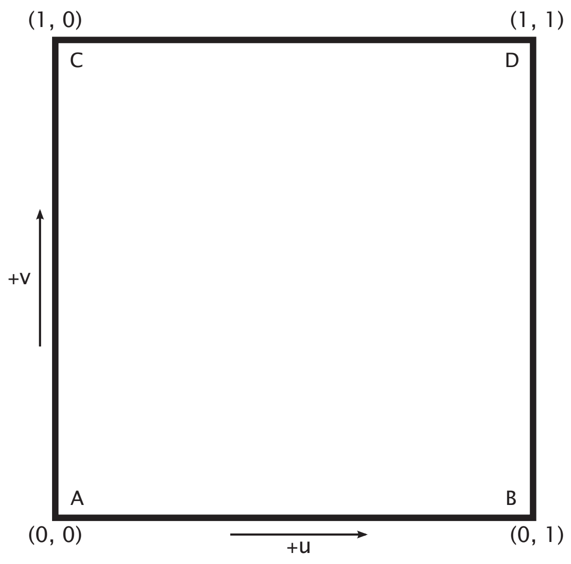

Figure 8.25: Parameterization of a quad

Domain parameterization of a quad is two-dimensional and can be represented as a two-dimensional vector. This can be smoothly interpolated over the quad to find the value of the vector at any point within it. For each of the quad's four vertices A, B, C, and D, the values of the vector will be $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$, respectively. We can generate these values per vertex in our geometry shader and pass them to the fragment shader.

To use this vector to retrieve the interpolated values of our other per-fragment attributes, we make the following observation: The value of any interpolant will move smoothly between vertex A and B and between C and D with the x component of the vector. Likewise, a value along the edge AB will move smoothly to the corresponding value on edge CD. Thus, given the values of the attributes at the vertices A through D, we can use the domain parameter to interpolate a value of each attribute at any point inside the quad.

Thus, our geometry shader simply passes all four of the per-vertex attributes, unmodified, as **flat** outputs to the fragment shader, along with a smoothly varying domain parameter per vertex. The fragment shader then uses the domain parameter and *all four* per-vertex attributes to perform the interpolation directly.

The geometry shader is shown in Listing 8.34, and the fragment shader is shown in Listing 8.35 — both are taken from the gsquads example. Finally, the result of rendering the same geometry as shown in Figure 8.24 is shown in Figure 8.26.

```
#version 430 core

layout (lines_adjacency) in;
layout (triangle_strip, max_vertices = 6) out;

in VS_OUT
{
    vec4 color;
} gs_in[4];

out GS_OUT
{
    flat vec4 color[4];
    vec2 uv;
} gs_out;

void main(void)
{
    gl_Position = gl_in[0].gl_Position;
    gs_out.uv = vec2(0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[1].gl_Position;
    gs_out.uv = vec2(1.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[2].gl_Position;
    gs_out.uv = vec2(1.0, 1.0);

    // We're only writing the output color for the last
    // vertex here because they're flat attributes,
    // and the last vertex is the provoking vertex by default
    gs_out.color[0] = gs_in[1].color;
    gs_out.color[1] = gs_in[0].color;
    gs_out.color[2] = gs_in[2].color;
    gs_out.color[3] = gs_in[3].color;
    EmitVertex();
    EndPrimitive();

    gl_Position = gl_in[0].gl_Position;
    gs_out.uv = vec2(0.0, 0.0);
    EmitVertex();

    gl_Position = gl_in[2].gl_Position;
    gs_out.uv = vec2(1.0, 1.0);
    EmitVertex();

    gl_Position = gl_in[3].gl_Position;
    gs_out.uv = vec2(0.0, 1.0);

    // Again, only write the output color for the last vertex
    gs_out.color[0] = gs_in[1].color;
    gs_out.color[1] = gs_in[0].color;
    gs_out.color[2] = gs_in[2].color;
    gs_out.color[3] = gs_in[3].color;
    EmitVertex();

    EndPrimitive();
}
```

Listing 8.34: Geometry shader for rendering quads

```
#version 430 core

in GS_OUT
{
    flat vec4 color[4];
    vec2 uv;
} fs_in;

out vec4 color;

void main(void)
{
    vec4 c1 = mix(fs_in.color[0], fs_in.color[1], fs_in.uv.x);
    vec4 c2 = mix(fs_in.color[2], fs_in.color[3], fs_in.uv.x);

    color = mix(c1, c2, fs_in.uv.y);
}
```

Listing 8.35: Fragment shader for rendering quads

Multiple Viewport Transformations

You learned in "Viewport Transformation" back in Chapter 3 about the viewport transformation and how you can specify the rectangle of the window you're rendering into by calling **glViewport()** and **glDepthRange()**. Normally, you would set the viewport dimensions to cover the entire window or screen, depending on whether your application is running on a desktop or is taking over the whole display. However, it's possible to move the viewport around and draw into

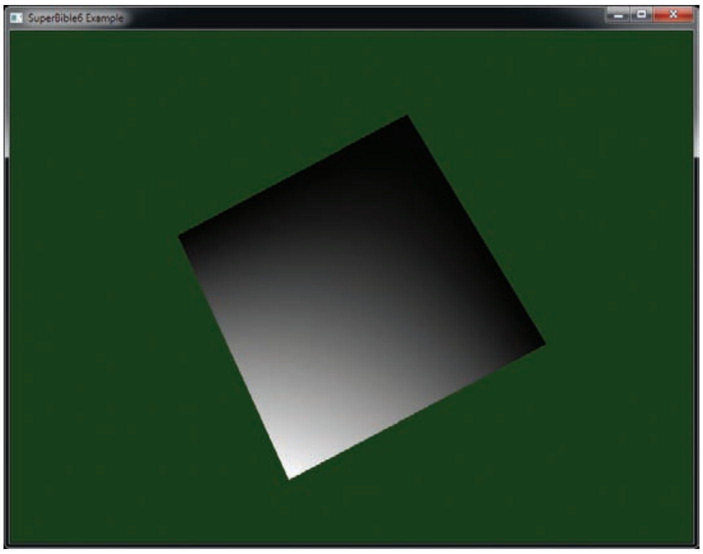

Figure 8.26: Quad rendered using a geometry shader

multiple virtual windows within a single larger framebuffer. Furthermore, OpenGL also allows you to use multiple viewports *at the same time*. This feature is known as viewport arrays.

To use a viewport array, we first need to tell OpenGL what the bounds of the viewports we want to use are. To do this, call **`glViewportIndexedf()`** or **`glViewportIndexedfv()`**, whose prototypes are

```
void glViewportIndexedf(GLuint index,
                        GLfloat x,
                        GLfloat y,
                        GLfloat w,
                        GLfloat h);

void glViewportIndexedfv(GLuint index,
                         const GLfloat * v);
```

For both **`glViewportIndexedf()`** and **`glViewportIndexedfv()`**, `index` is the index of the viewport you wish to modify. Also notice that the viewport parameters to the indexed viewport commands are floating-point values rather than the integers used for **`glViewport()`**. OpenGL supports a minimum[6] of 16 viewports, and so `index` can range from 0 to 15.

6. The actual number of viewports that are supported by OpenGL can be determined by querying the value of `GL_MAX_VIEWPORTS`.

Likewise, each viewport also has its own depth range, which can be specified by calling **glDepthRangeIndexed()**, whose prototype is

```
void glDepthRangeIndexed(GLuint index,
                         GLdouble n,
                         GLdouble f);
```

Again, index may be between 0 and 15. In fact, **glViewport()** really sets the extent of all of the viewports to the same range, and **glDepthRange()** sets the depth range of all viewports to the same range. If you want to set more than one or two of the viewports at a time, you might consider using **glViewportArrayv()** and **glDepthRangeArrayv()**, whose prototypes are

```
void glViewportArrayv(GLuint first,
                      GLsizei count,
                      const GLfloat * v);

void glDepthRangeArrayv(GLuint first,
                        GLsizei count,
                        const GLdouble * v);
```

These functions set either the viewport extents or depth range for count viewports starting with the viewport indexed by first to the parameters specified in the array v. For **glViewportArrayv()**, the array contains a sequence of x, y, width, height values, in that order. For **glDepthRangeArrayv()**, the array contains a sequence of n, f pairs, in that order.

Once you have specified your viewports, you need to direct geometry into them. This is done by using a geometry shader. Writing to the built-in variable gl_ViewportIndex selects the viewport to render into.
Listing 8.36 shows what such a geometry shader might look like.

```
#version 430 core

layout (triangles, invocations = 4) in;
layout (triangle_strip, max_vertices = 3) out;

layout (std140, binding = 0) uniform transform_block
{
    mat4 mvp_matrix[4];
};

in VS_OUT
{
    vec4 color;
} gs_in[];

out GS_OUT
{
    vec4 color;
} gs_out;
```

```
void main(void)
{
    for (int i = 0; i < gl_in.length(); i++)
    {
        gs_out.color = gs_in[i].color;
        gl_Position = mvp_matrix[gl_InvocationID] *
                      gl_in[i].gl_Position;
        gl_ViewportIndex = gl_InvocationID;
        EmitVertex();
    }
    EndPrimitive();
}
```

Listing 8.36: Rendering to multiple viewports in a geometry shader

When the shader of Listing 8.36 executes, it produces four invocations of the shader. On each invocation, it sets the value of `gl_ViewportIndex` to the value of `gl_InvocationID`, directing the result of each of the geometry shader instances to a separate viewport. Also, for each invocation, it uses a separate model-view-projection matrix, which it retrieves from the uniform block, `transform_block`. Of course, a more complex shader could be constructed, but this is sufficient to demonstrate direction of transformed geometry into a number of different viewports. We have implemented this code in the `multipleviewport` sample, and the result of running this shader on our simple spinning cube is shown in Figure 8.27.

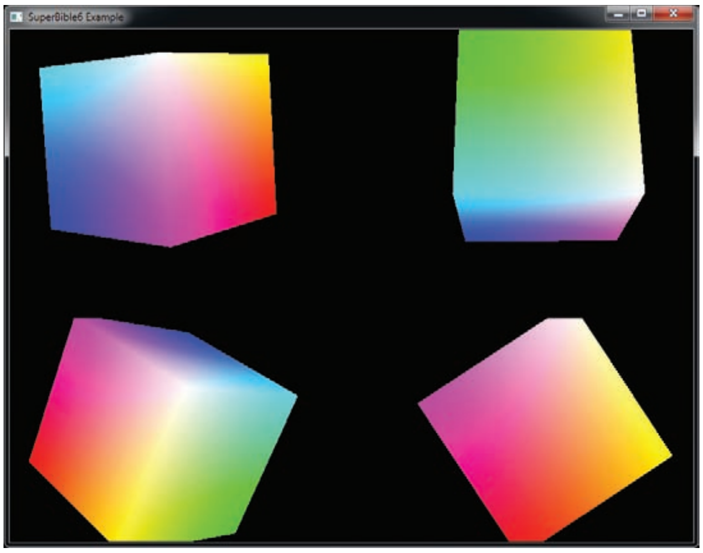

Figure 8.27: Result of rendering to multiple viewports

You can clearly see the four copies of the cube rendered by Listing 8.36 in Figure 8.27. Because each was rendered into its own viewport, it is clipped separately, and so where the cubes extend past the edges of their respective viewports, their corners are cut off by OpenGL's clipping stage.

Summary

In this chapter, you have read about the two tessellation shader stages, the fixed-function tessellation engine, and the way they interact. You have also read about geometry shaders and have seen how both the tessellator and the geometry shader can be used to change the amount of data in the OpenGL pipeline. You have also seen some of the additional functionality in OpenGL that can be accessed using tessellation and geometry shaders. You have seen how, conceptually, tessellation shaders and geometry shaders process vertices in groups — in the case of tessellation shaders, those groups forming *patches*, and in the case of geometry shaders, those groups forming traditional primitives such as lines and triangles. You've seen the special *adjacency* primitive types accessible to geometry shaders. After the geometry shader ends, primitives are eventually sent to the rasterizer and then to per-fragment operations, which will be the subject of the next chapter.