Chapter 9. Tessellation Shaders

Chapter Objectives

After reading this chapter, you'll be able to do the following:

- Understand the differences between tessellation and vertex shaders.
- Identify the phases of processing that occur when using tessellation shaders.
- Recognize the various *tessellation domain*s and know which one best matches the type of geometry you need to generate.
- Initialize data and draw using the patch geometric primitive.

This chapter introduces OpenGL's tessellation shader stages. It has the following major sections:

- "Tessellation Shaders" provides an overview of how tessellation shaders work in OpenGL.
- "Tessellation Patches" introduces tessellation's rendering primitive, the patch.
- "Tessellation Control Shaders" explains the operation and purpose of the first tessellation shading.
- "Tessellation Evaluation Shaders" describes the second tessellation stage and how it operates.
- "A Tessellation Example: The Teapot" shows an example of rendering a teapot using tessellation shaders and Bézier patches.
- "Additional Tessellation Techniques" discusses some additional techniques that are enabled by tessellation shading.

Tessellation Shaders

Up to this point, only vertex shaders have been available for us to manipulate geometric primitives. While there are numerous graphics techniques you can do using vertex shaders, they do have their limitations. One limitation is that they can't create additional geometry during their execution. They really only update the data associated with the current vertex they are processing, and they can't even access the data of other vertices in the primitives.

To address those issues, the OpenGL pipeline contains several other shader stages that address those limitations. In this chapter, we introduce *tessellation shaders*, which, for example, can generate a mesh of triangles using a new geometric primitive type called a patch.

Tessellation shading adds two shading stages to the OpenGL pipeline to generate a mesh of geometric primitives. As compared to having to specify all of the lines or triangles to form your model as you do with vertex shading. With tessellation, you begin by specifying a patch, which is just an ordered list of vertices. When a patch is rendered, the *tessellation control shader* executes first, operating on your patch vertices and specifying how much geometry should be generated from your patch. Tessellation control shaders are optional, and we'll see what's required if you don't use one. After the tessellation control shader completes, the second shader, the *tessellation evaluation shader*, positions the vertices of the generated mesh using *tessellation coordinates* and sends them to the rasterizer or to a geometry shader for more processing (which we describe in Chapter 10, "Geometry Shaders").

As we describe OpenGL's process of tessellation, we'll start at the beginning with describing patches in "Tessellation Patches" (next) and then move to describe the tessellation control shader's

operation detail in "Tessellation Control Shaders" on page 500. OpenGL passes the output of the tessellation control shader to the *primitive generator*, which generates the mesh of geometric primitives and Tessellation coordinates that the tessellation evaluation shader stage uses. Finally, the tessellation evaluation shader positions each of the vertices in the final mesh, a process described in "Tessellation Evaluation Shaders" on page 508.

We conclude the chapter with a few examples, including a demonstration of *displacement mapping*, which combines texture mapping for vertices (which is discussed in Chapter 6, "Textures and Framebuffers") with tessellation shaders.

Tessellation Patches

The tessellation process doesn't operate on OpenGL's classic geometric primitives—points, lines, and triangles—but uses a new primitive (added in OpenGL Version 4.0) called a *patch*. Patches are processed by all active shading stages in the pipeline. By comparison, other primitive types are processed only by vertex, fragment, and geometry shaders, and bypass the tessellation stage. In fact, if any tessellation shaders are active, passing any other type of geometry will generate a **GL_INVALID_OPERATION** error. Conversely, you'll get a **GL_INVALID_OPERATION** error if you try to render a patch without any tessellation shaders bound (specifically, a tessellation evaluation shader; we'll see that tessellation control shaders are optional).

Patches are nothing more than lists of vertices that you pass into OpenGL, which preserves their order during processing. When rendering with tessellation and patches, you use OpenGL rendering commands, like **glDrawArrays()**, and specify the total number of vertices to be read from the bound vertex-buffer objects and processed for that draw call. When you're rendering with the other OpenGL primitives, OpenGL implicitly knows how many vertices to use based on the primitive type you specified in your draw call, like using three vertices to make a triangle. However, when you use a patch, OpenGL needs to be told how many vertices from your vertex array to use to make one patch, which you specify using **glPatchParameteri()**. Patches processed by the same draw call will all be the same size.

void **glPatchParameteri**(GLenum *pname*, GLint *value*);

Specifies the number of vertices in a patch using *value*. *pname* must be set to **GL_PATCH_VERTICES**.

A **GL_INVALID_ENUM** error is generated if *value* is less than zero or greater than **GL_MAX_PATCH_VERTICES**.

The default number of vertices for a patch is three. If the number of vertices for a patch is less that *value*, the patch is ignored, and no geometry will be generated.

To specify a patch, use the input type **GL_PATCHES** into any OpenGL drawing command. Example 9.1 demonstrates issuing two patches, each with four vertices.

Example 9.1 Specifying Tessellation Patches

Click here to view code image

```
GLfloat vertices [][2] = {
  {-0.75, -0.25}, {-0.25, -0.25}, {-0.25, 0.25}, {-0.75, 0.25},
  { 0.25, -0.25}, { 0.75, -0.25}, { 0.75, 0.25}, { 0.25, 0.25}
};

glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
             GL_STATIC_DRAW);

glVertexAttribPointer(vPos, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));
glPatchParameteri(GL_PATCH_VERTICES, 4);
glDrawArrays(GL_PATCHES, 0, 8);
```

The vertices of each patch are first processed by the currently bound vertex shader and then used to initialize the array `gl_in`, which is implicitly declared in the tessellation control shader. The number of elements in `gl_in` is the same as the patch size specified by **glPatchParameteri()**. Inside a tessellation control shader, the variable `gl_PatchVerticesIn` provides the number of elements in `gl_in` (as does querying `gl_in.length()`).

Tessellation Control Shaders

Once your application issues a patch, the tessellation control shader will be called (if one is bound) and is responsible for completing the following actions:

- Generate the *tessellation output patch vertices* that are passed to the tessellation evaluation shader, as well as update any per-vertex or per-patch attribute values as necessary.
- Specify the *tessellation level factor*s that control the operation of the primitive generator. These are special tessellation control shader variables called `gl_TessLevelInner` and `gl_TessLevelOuter`, and are implicitly declared in your tessellation control shader.

We discuss each of these actions in turn.

Generating Output-Patch Vertices

Tessellation control shaders use the vertices specified by the application, which we'll call *input-patch vertices*, to generate a new set of vertices, the output-patch vertices, which are stored in the `gl_out` array of the tessellation control shader. At this point, you might be asking what's going on; why not just pass in the original set of vertices from the application and skip all this work? Tessellation control shaders can modify the values passed from the application, but they can also create or remove vertices from the input-patch vertices when producing the output-patch vertices. You might use this functionality when working with sprites or when minimizing the amount of data sent from the application to OpenGL, which may increase performance.

You already know how to set the number of input-patch vertices using **glPatchParameteri()**. You specify the number of output-patch vertices using a **layout** construct in your tessellation control shader, which sets the number of output-patch vertices to 16:

```
layout (vertices = 16) out;
```

The value set by the `vertices` parameter in the **layout** directive does two things: It sets the size of the output-patch vertices, `gl_out`; and it specifies how many times the tessellation control shader will execute (once for each output-patch vertex).

In order to determine which output vertex is being processed, the tessellation control shader can use the `gl_InvocationID` variable. Its value is most often used as an index into the `gl_out` array. While a tessellation control shader is executing, it has access to all patch vertex data—both input and output. This can lead to issues where a shader invocation might need data values from a shader invocation that hasn't happened yet. Tessellation control shaders can use the GLSL `barrier()` function, which causes all of the control shaders for an input patch to execute and wait until all of them have reached that point, thus guaranteeing that all of the data values you might set will be computed.

A common idiom of tessellation control shaders is just passing the input-patch vertices out of the shader. Example 9.2 demonstrates this for an output patch with four vertices.

Example 9.2 Passing Through Tessellation Control Shader Patch Vertices

Click here to view code image

```
#version 420 core

layout (vertices = 4) out;

void
main()
{
    gl_out[gl_InvocationID].gl_Position
                        = gl_in[gl_InvocationID].gl_Position;

    // and then set tessellation levels
}
```

Tessellation Control Shader Variables

The **gl_in** array is actually an array of structures, with each element defined as

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

For each value that you need downstream (e.g., in the tessellation evaluation shader), you'll need to assign values similar to what we did with the `gl_Position` field.

The `gl_out` array has the same fields but is a different size specified by `gl_PatchVerticesOut`, which, as we saw, was set in the tessellation control shader's **out layout** qualifier. Additionally, the scalar values shown in Table 9.1 are provided for determining which primitive and output vertex invocation is being shaded.

Variable Declaration	Description
`gl_InvocationID`	Invocation index for the output vertex of the current tessellation control shader
`gl_PrimitiveID`	Primitive index for current input patch
`gl_PatchVerticesIn`	Number of vertices in the input patch, which is the dimension of `gl_in`
`gl_PatchVerticesOut`	Number of vertices in the output patch, which is the dimension of `gl_out`

Table 9.1 Tessellation Control Shader Input Variables

If you have additional per-vertex attribute values, either for input or output, these need to be declared as either **in** or **out** arrays in your tessellation control shader. The size of an input array needs to be sized to the input-patch size or can be declared unsized, and OpenGL will appropriately allocate space for all its values. Similarly, per-vertex output attributes, which you will be able to access in the tessellation evaluation shader, need to be sized to the number of vertices in the output patch or can be declared unsized as well.

Controlling Tessellation

The other function of a tessellation control shader is to specify how much to tessellate the output patch. While we haven't discussed tessellation evaluation shaders in detail yet, they control the type of output patch for rendering and, consequently, the domain where tessellation occurs. OpenGL supports three tessellation domains: a quadrilateral, a triangle, and a collection of isolines.

The amount of tessellation is controlled by specifying two sets of values: the inner and outer tessellation levels. The outer tessellation levels control how the perimeter of the domain is subdivided and are stored in an implicitly declared four-element array named `gl_TessLevelOuter`. Similarly, the inner tessellation levels specify how the interior of the domain is subdivided and are stored in a two-element array named `gl_TessLevelInner`. All tessellation level factors are floating-point values, and we'll see the effect that fractional values have on tessellations in a bit. One final point is that while the dimensions of the implicitly declared tessellation level factors arrays are fixed, the number of values used from those arrays depends on the type of tessellation domain.

Understanding how the inner and outer tessellation levels operate is key to getting tessellation to do what you want. Each of the tessellation level factors specifies how many "segments" to subdivide a region, as well as how many tessellation coordinates and geometric primitives to generate. How that subdivision is done varies by domain type. We discuss each type of domain in turn, as each domain type operates differently.

Quad Tessellation

Using the quadrilateral domain may be the most intuitive, so we begin with it. It's useful when your input patches are rectangular, as you might have when using two-dimensional spline surfaces, like Bézier surfaces. The quad domain subdivides the *unit square* using all of the inner and outer tessellation levels. For instance, if we were to set the tessellation level factors to the following values, OpenGL would tessellate the quad domain, as illustrated in Figure 9.1.

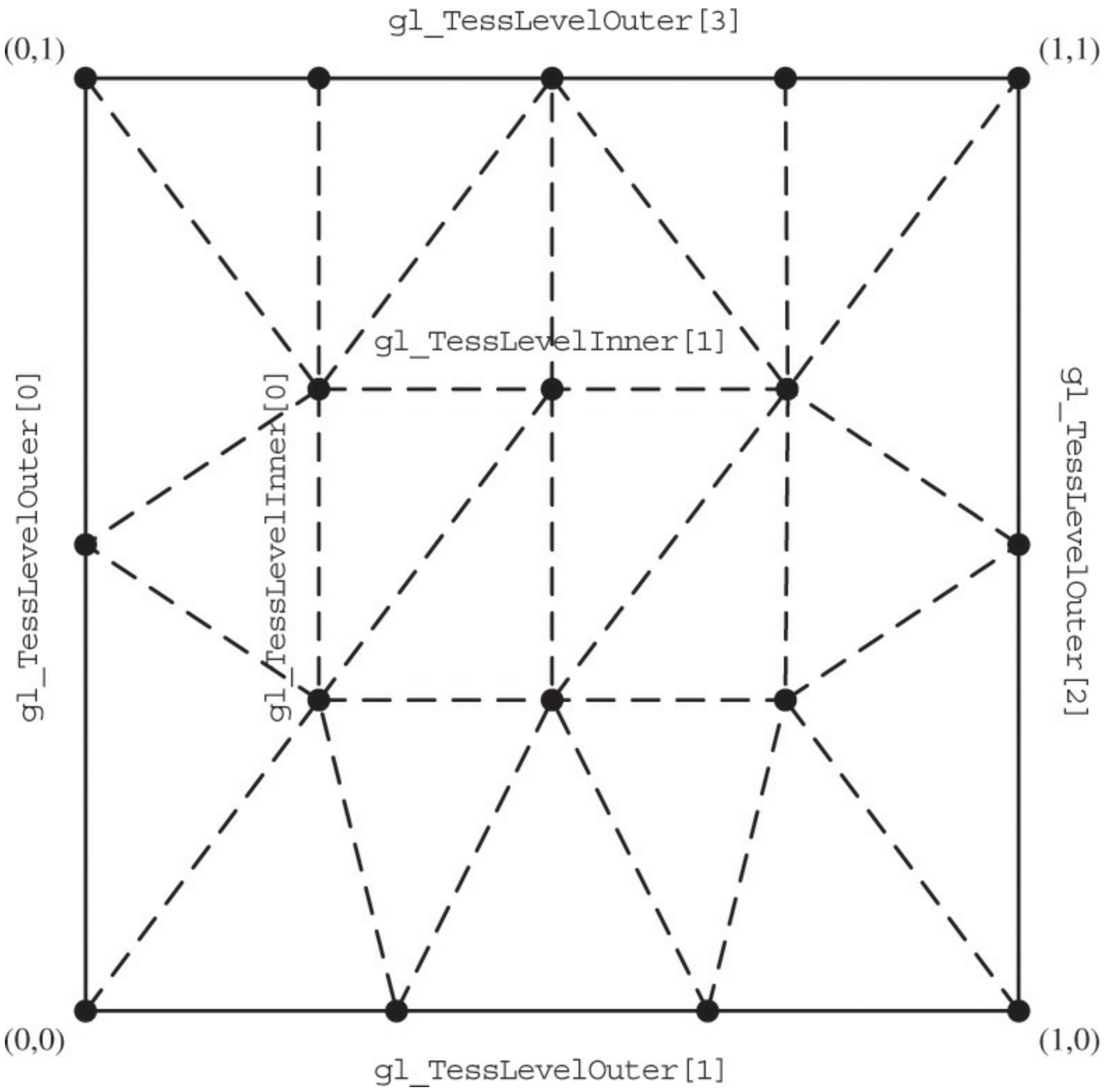

Figure 9.1 Quad tessellation
A tessellation of a quad domain using the tessellation levels from Example 9.3.

Example 9.3 Tessellation Levels for Quad Domain Tessellation Illustrated in Figure 9.1

```
gl_TessLevelOuter[0] = 2.0;
gl_TessLevelOuter[1] = 3.0;
gl_TessLevelOuter[2] = 2.0;
gl_TessLevelOuter[3] = 5.0;

gl_TessLevelInner[0] = 3.0;
gl_TessLevelInner[1] = 4.0;
```

Notice that the outer tessellation level values correspond to the number of segments for each edge

around the perimeter, while the inner tessellation levels specify how many "regions" are in the horizontal and vertical directions in the interior of the domain. Also shown in Figure 9.1 is a possible triangularization of the domain,[1] shown using the dashed lines. Likewise, the solid circles represent the tessellation coordinates, each of which will be provided as input into the tessellation evaluation shader. In the case of the quad domain, the tessellation coordinates will have two coordinates, (u, v), which will both be in the range [0, 1], and each tessellation coordinate will be passed into an invocation of an tessellation evaluation shader.

1. Triangularization of the domain is implementation-dependent.

Isoline Tessellation

Similar to the quad domain, the isoline domain also generates (u, v) pairs as tessellation coordinates for the tessellation evaluation shader. Isolines, however, use only two of the outer tessellation levels to determine the amount of subdivision (and none of the inner tessellation levels). This is illustrated in Figure 9.2 for the tessellation level factors shown in Example 9.4.

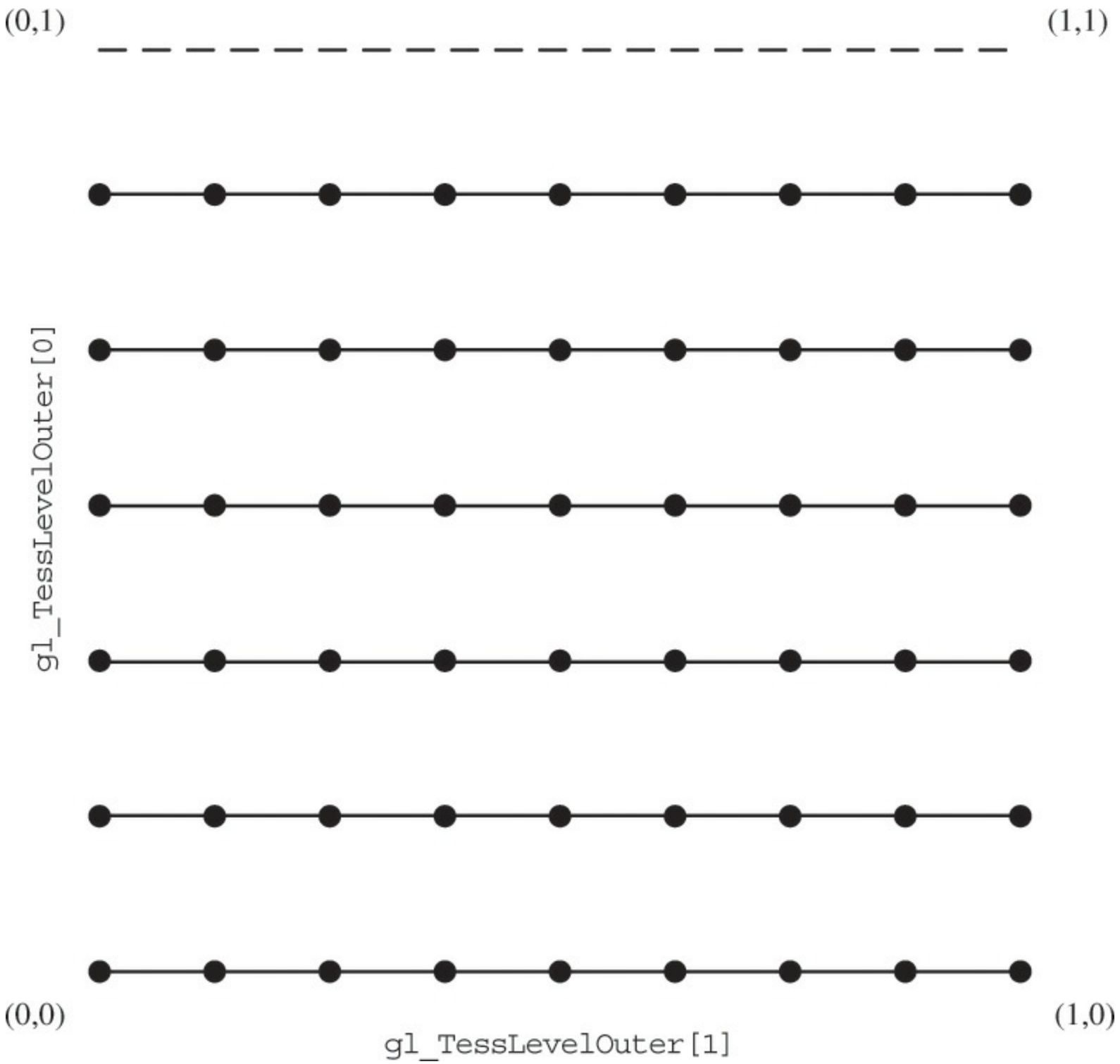

Figure 9.2 Isoline tessellation
A tessellation of an isolines domain using the tessellation levels from Example 9.4.

Example 9.4 Tessellation Levels for an Isoline Domain Tessellation Shown in Figure 9.2

```
gl_TessLevelOuter[0] = 6;
gl_TessLevelOuter[1] = 8;
```

Notice the dashed line along the $v = 1$ edge. That's because isolines don't include a tessellated isoline along that edge, and if you place two isoline patches together (i.e., they share an edge of two patches), there isn't overlap of the edges.

Triangle Tessellation

Finally, let's discuss tessellation using a triangle domain. As compared to either the quad or isolines domains, coordinates related to the three vertices of a triangle aren't very conveniently represented by a (u, v) pair. Instead, triangular domains use *barycentric coordinates* to specify their tessellation coordinates. Barycentric coordinates are represented by a triplet of numbers, (a, b, c), each of which lies in the range [0, 1], and which have the property that $a + b + c = 1$. Think of a, b, and c as weights for each individual triangle vertex.

As with any of the other domains, the generated tessellation coordinates are a function of the tessellation level factors. In particular, the first three outer tessellation levels and only inner tessellation level zero. The tessellation of a triangular domain with tessellation level factors set as in Example 9.5 is shown in Figure 9.3.

Example 9.5 Tessellation Levels for a Triangular Domain Tessellation Shown in Figure 9.3.

```
gl_TessLevelOuter[0] = 6;
gl_TessLevelOuter[1] = 5;
gl_TessLevelOuter[2] = 8;

gl_TessLevelInner[0] = 5;
```

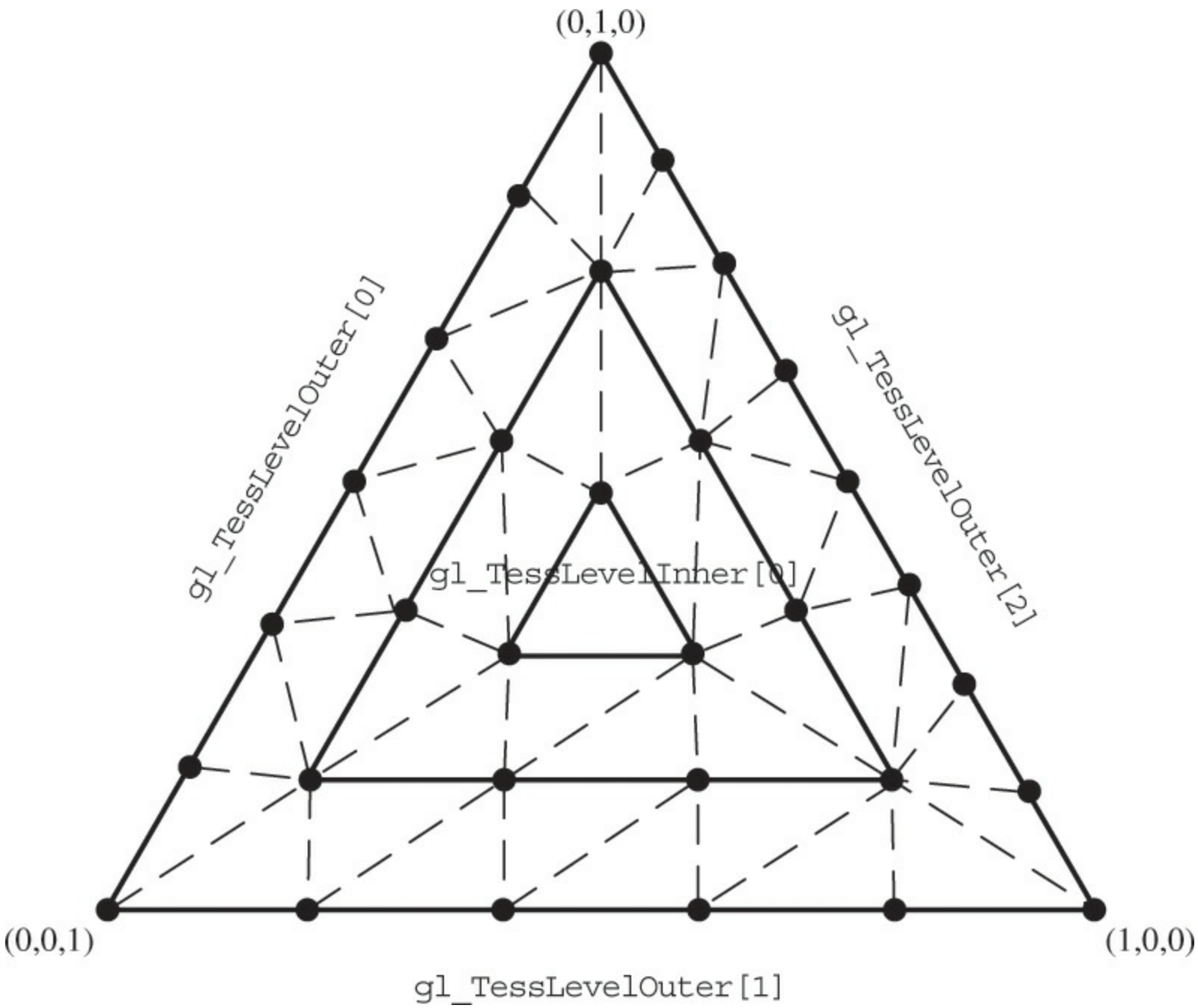

Figure 9.3 Triangle tessellation

A tessellation of a triangular domain using the tessellation levels from Example 9.5.

As with the other domains, the outer tessellation level control the subdivision of the perimeter of the triangle and the inner tessellation level controls how the interior is partitioned. As compared to the rectangular domains, where the interior is partitioned in a set of rectangles forming a grid, the interior of the triangular domain is partitioned into a set of concentric triangles that form the regions. Specifically, let t represent the inner tessellation level. If t is an even value, then the center of the triangular domain—barycentric coordinate $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$—is located, and then $(t/2) - 1$ concentric triangles are generated between the center point and the perimeter. Conversely, if t is an odd value, then $(t/2) - 1$ concentric triangles are generated to the perimeter, and the center point (in barycentric coordinates) will not be a tessellation coordinate. These two scenarios are shown in Figure 9.4.

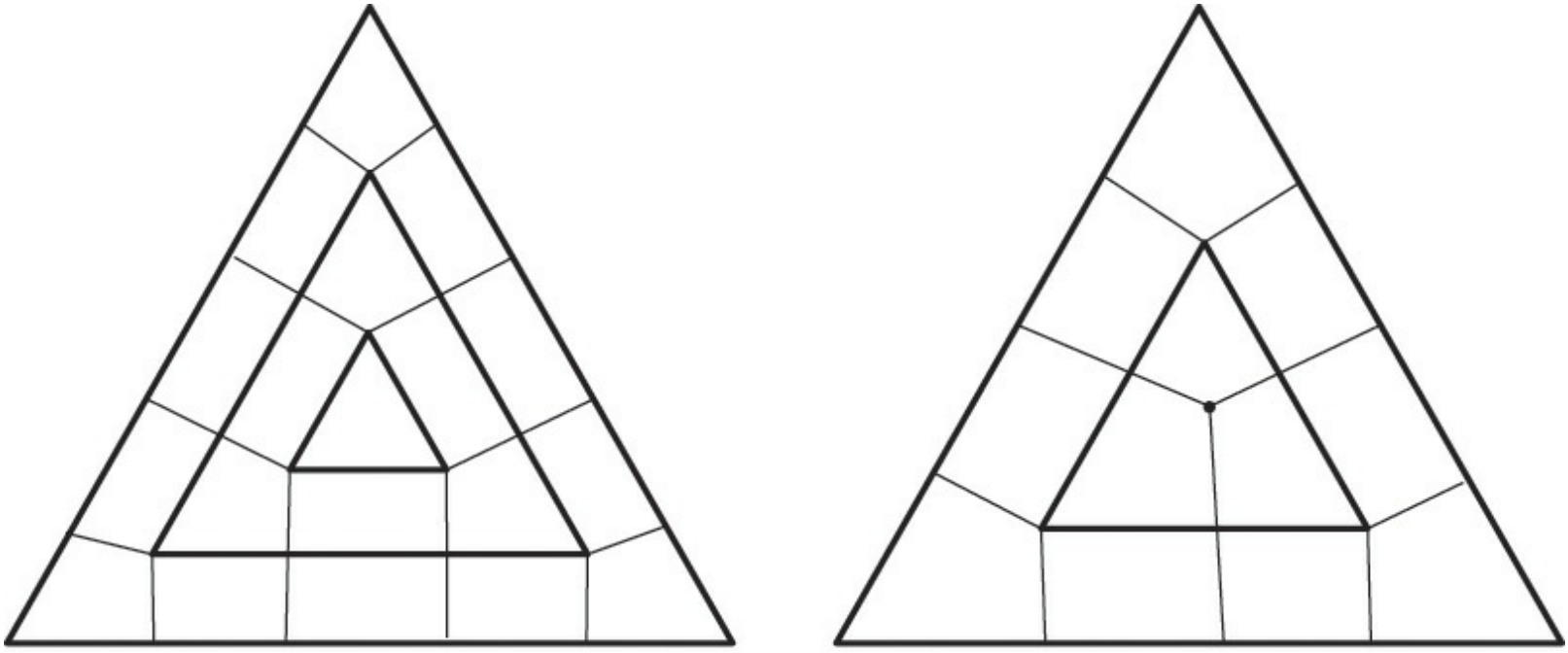

Figure 9.4 Even and odd tessellation
Examples of how even and odd inner tessellation levels affect triangular tessellation.

Bypassing the Tessellation Control Shader

As we mentioned, often, your tessellation control shader will be just a pass-through shader, copying data from input to output. In such a case, you can actually bypass using a tessellation control shader and set the tessellation level factors using the OpenGL API, as compared to using a shader. The **glPatchParameterfv()** function can be used to set the inner and outer tessellation levels.

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

Sets the inner and outer tessellation levels for when no tessellation control shader is bound. *pname* must be either **GL_PATCH_DEFAULT_OUTER_LEVEL** or **GL_PATCH_DEFAULT_INNER_LEVEL**.

When *pname* is **GL_PATCH_DEFAULT_OUTER_LEVEL**, values must be an array of four floating-point values that specify the four outer tessellation levels.

Similarly, when *pname* is **GL_PATCH_DEFAULT_INNER_LEVEL**, values must be an array of two floating-point values that specify the two inner tessellation levels.

Tessellation Evaluation Shaders

The final phase in OpenGL's tessellation pipeline is the tessellation evaluation shader execution. The bound tessellation evaluation shader is executed one for each tessellation coordinate that the

primitive generator emits and is responsible for determining the position of the vertex derived from the tessellation coordinate. As we'll see, tessellation evaluation shaders look similar to vertex shaders in transforming vertices into screen positions (unless the tessellation evaluation shader's data is going to be further processed by a geometry shader).

The first step in configuring a tessellation evaluation shader is to configure the primitive generator, which is done using a **layout** directive, similar to what we did in the tessellation control shader. Its parameters specify the tessellation domain and, subsequently, the type of primitives generated; face orientation for solid primitives (used for face culling); and how the tessellation levels should be applied during primitive generation.

Specifying the Primitive Generation Domain

We now describe the parameters that you will use to set up the tessellation evaluation shader's **out layout** directive. First, we talk about specifying the tessellation domain. As you've seen, three types of domains are used for generating tessellation coordinates, which are described in Table 9.2.

Primitive Type	Description	Domain Coordinates
`quads`	A rectangular domain over the unit square	a (u, v) pair with u, v values ranging from 0 to 1.
`triangles`	A triangular shaped domain using barycentric coordinates	(a, b, c) with a, b, and c values ranging from 0 to 1 and where $a + b + c = 1$
`isolines`	A collection of lines across the unit square	a (u, v) pair with u values ranging from 0 to 1, and v values ranging from 0 to almost 1

Table 9.2 Evaluation Shader Primitive Types

Specifying the Face Winding for Generated Primitives

As with any filled primitive in OpenGL, the order the vertices are issued determines the face-ness of the primitive. Because we don't issue the vertices directly in this case, but have the primitive generator do it on our behalf, we need to tell it the face winding of our primitives. In the **layout** directive, specify `cw` for clockwise *vertex winding* or `ccw` for counterclockwise vertex winding.

Specifying the Spacing of Tessellation Coordinates

Additionally, we can control how fractional values for the outer tessellation levels are used in determining the tessellation coordinate generation for the perimeter edges. (Inner tessellation levels are affected by these options.) Table 9.3 describes the three spacing options available, where *max* represents an OpenGL implementation's maximum accepted value for a tessellation level.

Option	Description
`equal_spacing`	Tessellation level is clamped to [1, *max*] and then rounded up to the next-largest integer value.
`fractional_even_spacing`	The value is clamped to [2, *max*] and then rounded up to the next-largest even integer value *n*. The edge is then divided into $n-2$ equal length parts and two other parts, one at either end, that may be shorter than the other lengths.
`fractional_odd_spacing`	The value is clamped to [1, *max* − 1] and then rounded up to the next-largest odd integer value *n*. The edge is then divided into $n-2$ equal length parts and two other parts, one at either end, that may be shorter than the other lengths.

Table 9.3 Options for Controlling Tessellation Level Effects

Additional Tessellation Evaluation Shader `layout` Options

Finally, should you want to output points, as compared to isolines or filled regions, you can supply the `point_mode` option, which will render a single point for each vertex processed by the tessellation evaluation shader.

The order of options within the `layout` directive is not important. As an example, the following `layout` directive will request primitives generated on a triangular domain using equal spacing, counterclockwise-oriented triangles, and only rendering points as compared to connected primitives.

Click here to view code image

```
layout (triangles, equal_spacing, ccw, points) out;
```

Specifying a Vertex's Position

The vertices output from the tessellation control shader (i.e., the `gl_Position` values in `gl_out` array) are made available in the evaluation shader in the `gl_in` variable, which, when combined with tessellation coordinates, can be used to generate the output vertex's position.

Tessellation coordinates are provided to the shader in the variable `gl_TessCoord`. In Example 9.6, we use a combination of equal-spaced quads to render a simple patch. In this case, the tessellation coordinates are used to color the surface and illustrate how to compute the vertex's position.

Example 9.6 A Sample Tessellation Evaluation Shader

Click here to view code image

```
#version 420 core

layout (quads, equal_spacing, ccw) in;

out vec4 color;

void
```

```
main()
{
    float u = gl_TessCoord.x;
    float omu = 1 - u; // one minus 'u'
    float v = gl_TessCoord.y;
    float omv = 1 - v; // one minus 'v'

    color = gl_TessCoord;

    gl_Position =
      omu * omv * gl_in[0].gl_Position +
      u   * omv * gl_in[1].gl_Position +
      u   *   v * gl_in[2].gl_Position +
      omu *   v * gl_in[3].gl_Position;
}
```

Tessellation Evaluation Shader Variables

Similar to tessellation control shaders, tessellation evaluation shaders have a `gl_in` array that is actually an array of structures, with each element defined as shown in Example 9.7.

Example 9.7 `gl_in` Parameters for Tessellation Evaluation Shaders

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

Additionally, the scalar values in Table 9.4 are provided for determining the current primitive and the position of the output vertex.

Variable Declaration	Description
`gl_PrimitiveID`	Primitive index for current input patch
`gl_PatchVerticesIn`	Number of vertices in the input patch, which is the dimension of `gl_in`
`gl_TessLevelOuter[4]`	Outer tessellation level values
`gl_TessLevelInner[2]`	Inner tessellation level values
`gl_TessCoord`	Coordinates in patch domain space of the vertex being shaded in the evaluation shader

Table 9.4 Tessellation Control Shader Input Variables

The output vertex's data is stored in an interface block defined as follows:

```
out gl_PerVertex {
    vec4 gl_Position;
    float gl_PointSize;
    float gl_ClipDistance[];
    float gl_CullDistance[];
};
```

A Tessellation Example: The Teapot

All of that theory could use a concrete demonstration. In this section, we render the famous *Utah teapot* using Bézier patches. A Bézier patch, named after French engineer Pierre Bézier, defines a parametric surface evaluated over the unit square using control points arranged in a grid. For our example, we use 16 control points arranged in a 4 × 4 grid. As such, we make the following observations to help us set up our tessellation:

- Bézier patches are defined over the unit square, which indicates we should use the `quads` domain type, specified in our **layout** directive in the tessellation evaluation shader.
- Each patch has 16 control points, so our **GL_PATCH_VERTICES** should be set to 16 using **glPatchParameteri()**.
- The 16 control points also define the number of input-patch vertices, which tells us our maximum index into the `gl_in` array in our tessellation control shader.
- Finally, because the tessellation control shader doesn't add any vertices to or remove any vertices from the patch, the number of output-patch vertices will also be 16, which specifies the value we use in our **layout** directive in the tessellation control shader.

Processing Patch Input Vertices

Given the information from our patches, we can easily construct the tessellation control shader for our application, which is shown in Example 9.8.

Example 9.8 Tessellation Control Shader for Teapot Example

Click here to view code image

```
#version 420 core

layout (vertices = 16) out;

void
main()
{
    gl_TessLevelInner[0] = 4;
    gl_TessLevelInner[1] = 4;

    gl_TessLevelOuter[0] = 4;
    gl_TessLevelOuter[1] = 4;
    gl_TessLevelOuter[2] = 4;
    gl_TessLevelOuter[3] = 4;

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

Using the tessellation level factors from Example 9.8, Figure 9.5 shows the patches of the teapot (shrunk slightly to expose each individual patch).

Figure 9.5 The tessellated patches of the teapot

This is a simple example of a tessellation control shader. In fact, it's a great example of a pass-through shader, where mostly the inputs are copied to the output. The shader also sets the inner and outer tessellation levels to constant values, which could also be done in the application using a call to **glPatchParameterfv()**. However, we include the example here for completeness.

Evaluating Tessellation Coordinates for the Teapot

Bézier patches use a bit of mathematics to determine the final vertex position from the input control points. The equation mapping a tessellation coordinate to a vertex position for our 4 × 4 patch is

$$\vec{p}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B(i,u)B(j,v)\vec{v}_{ij}$$

where $\vec{p}$ is the final vertex position, $\vec{v}_{ij}$ is the input control point at index (i, j) in our input patch (both of which are **vec4** s in GLSL), and the two occurrences of B are scaling functions.

Although it might not seem like it, we can map easily the formula to an tessellation evaluation shader, as shown in Example 9.9. In the following shader, the B function will be a GLSL function we'll define in a moment.

We also specify our `quads` domain, spacing options, and polygon face orientation in the **layout** directive.

Example 9.9 The `main` Routine of the Teapot Tessellation Evaluation Shader

Click here to view code image

```
#version 420 core

layout (quads, equal_spacing, ccw) out;

uniform mat4  MV;  // Model-view matrix
uniform mat4  P;   // Projection matrix

void
main()
{
    vec4  p = vec4(0.0);

    float  u = gl_TessCoord.x;
    float  v = gl_TessCoord.y;

    for (int j = 0; j < 4; ++j) {
        for (int i = 0; i < 4; ++i) {
            p += B(i, u) * B(j, v) * gl_in[4*j+i].gl_Position;
        }
    }

    gl_Position = P * MV * p;
}
```

Our B function is one of the *Bernstein polynomials*, which is an entire family of mathematical functions. Each one returns a scalar value. We're using a small, select set of functions, which we index using the first parameter, and we evaluate the function's value at one component of our tessellation coordinate. Here's the mathematical definition of our functions

$$B(i,u) = \binom{3}{i} u^i (1-u)^{3-i}$$

where the $\binom{3}{i}$ is a particular instance of a mathematical construct called a *binomial coefficient*.[2] We spare you the gory detail and just say we're lucky that it evaluates to either 1 or 3 in our cases, and which we hard-code into a lookup table, *bc* in the function's definition, and index using i. As such, we can rewrite $B(i, u)$ as

$$B(i,u) = \mathrm{bc}_i\, u^i (1-u)^{3-i}$$

2. Binomial coefficients in generally defined using the formula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, where $n!$ is the *factorial* of n, which is just the product of the values n to 1: $n! = (n)(n-1)(n-2)\ldots(2)(1)$.

This also translates easily into GLSL, as shown in Example 9.10.

Example 9.10 Definition of $B(i, u)$ for the Teapot Tessellation Evaluation Shader

Click here to view code image

```
float
B(int i, float u)
{
    // Binomial coefficient lookup table
    const vec4 bc = vec4(1, 3, 3, 1);
```

```
    return bc[i] * pow(u, i) * pow(1.0 - u, 3 - i);
}
```

While that conversation involved more mathematics than most of the other techniques we've described in the book, it is representative of what you will encounter when working with tessellated surfaces. While discussion of the mathematics of surfaces is outside the scope of this text, copious resources are available that describe the required techniques.

Additional Tessellation Techniques

In this final section, we briefly describe a few additional techniques you can employ while using tessellation shaders.

View-Dependent Tessellation

Most of the examples in this chapter have set the tessellation level factors to constant values (either in the shader or through uniform variables). One key feature of tessellation is being able to compute tessellation levels dynamically in the tessellation control shader and in particular basing the amount of tessellation on view-dependent parameters.

For example, you might implement a level-of-detail scheme based on the distance of the patch from the eye's location in the scene. In Example 9.11, we use the average of all the input-patch vertices to specify a single representative point for the patch and derive all the tessellation level factors from the distance of that point to the eye point.

Example 9.11 Computing Tessellation Levels Based on View-Dependent Parameters

Click here to view code image

```
uniform vec3 EyePosition;

void
main()
{
    vec4 center = vec4(0.0);

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

    center /= gl_in.length();

    float d = distance(center, vec4(EyePosition, 1.0));

    const float lodScale = 2.5;  // distance scaling variable

    float  tessLOD = mix(0.0, gl_MaxTessGenLevel, d * lodScale);
    for (int i = 0; i < 4; ++i) {
        gl_TessLevelOuter[i] = tessLOD;
    }

    tessLOD = clamp(0.5 * tessLOD, 0.0, gl_MaxTessGenLevel);
    gl_TessLevelInner[0] = tessLOD;
    gl_TessLevelInner[1] = tessLOD;
```

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

Example 9.11 is a very rudimentary method for computing a patch's level of detail. In particular, each perimeter edge is tessellated the same amount regardless of its distance from the eye. This doesn't take full advantage of tessellation possibilities based on view information, which is usually employed as a geometry optimization technique (i.e., reducing the object's geometric complexity the farther from the eye that object is, assuming that a perspective projection is used). Another failing of this approach is that if you have multiple patches that share edges, it's likely that the shared edges may be assigned different levels of tessellation depending on the objects orientation with respect to the eye's position, which might lead to *cracking* along the shared edges. Cracking is an important issue with tessellation, and we address another concern in "Shared Tessellated Edges and Cracking" on page 518.

To address guaranteeing that shared edges are tessellated the same, we need to find a method that returns the same tessellation factor for those edges. However, as compared to Example 9.11, which doesn't need to know anything about the logical ordering of input-patch vertices, any algorithm that needs to know which vertices are incident on a perimeter edge is data-dependent. This is because a patch is a logical ordering; only the application knows how it ordered the input-patch vertices. For Example 9.12, we introduce the following array of structures that contain our edge information for our tessellation control shader.

```
struct EdgeCenters {
    vec4 edgeCenter[4];
};
```

The application would need to populate this array using the world-space positions of the centers of each edge. In that example, we assume we're working with the `quads` domain, which is why there are four points in each `EdgeCenters` structure; the number of points would need to be modified for the other domains. The number of `EdgeCenters` structures in the array is the numbers of patches that will be issued in the draw call processed. We would modify the tessellation control shader to implement the following:

Example 9.12 Specifying Tessellation Level Factors Using Perimeter Edge Centers

Click here to view code image

```
struct EdgeCenters { vec4 edgeCenter[4]; };

uniform vec3  EyePosition;

uniform EdgeCenters patch[];

void
main()
{
    for (int i = 0; i < 4; ++i) {
        float d = distance(patch[gl_PrimitiveID].edgeCenter[i],
                           vec4(EyePosition, 1.0));

        const float lodScale = 2.5;  // distance scaling variable
```

```
            float tessLOD = mix(0.0, gl_MaxTessGenLevel, d * lodScale);

            gl_TessLevelOuter[i] = tessLOD;
        }

        tessLOD = clamp(0.5 * tessLOD, 0.0, gl_MaxTessGenLevel);
        gl_TessLevelInner[0] = tessLOD;
        gl_TessLevelInner[1] = tessLOD;

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

Shared Tessellated Edges and Cracking

Often, a geometric model that uses tessellation will have patches with shared edges. Tessellation in OpenGL guarantees that the geometry generated for the primitives within a patch won't have any cracks between them, but it can't make the same claim for patches that share edges. That's something the application needs to address, and clearly, the starting point is that shared edges need to be tessellated the same amounts. However, a secondary issue that can creep in: precision in mathematical computations done by a computer.

For all but trivial tessellation applications, the points along a perimeter edge will be positioned using multiple tessellation control shader output-patch vertices, which are combined with the tessellation coordinates in the tessellation evaluation shader. In order to truly prevent cracking along edges between similarly tessellated adjacent patches, the order of accumulation of mathematical operations in the tessellation evaluation shader must also match. Depending upon how the tessellation evaluation shader generates the mesh's vertices final positions, you may need to reorder the processing of vertices in the tessellation evaluation shader. A common approach to this problem is to recognize the output-patch vertices that contribute to a vertex incident to a perimeter edge and sort those vertices in a predictable manner, say, in terms of increasing magnitude along the edge.

Another technique to avoid cracking is applying the **precise** qualifier to shader computations where points might be in reversed order between two shader invocations. This is illustrated in Figure 9.6.

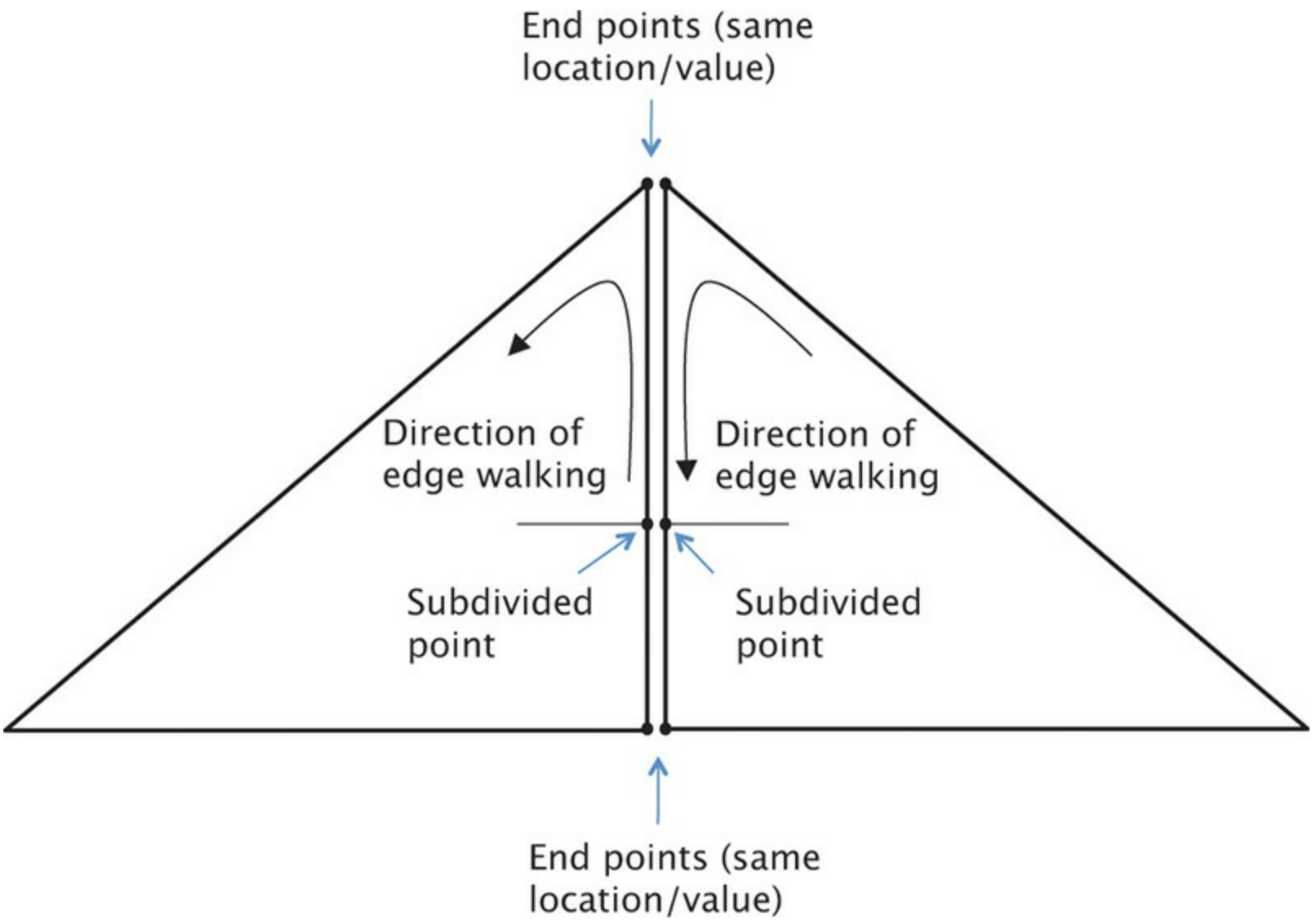

Figure 9.6 Tessellation cracking

When walking the interior edge in opposite directions, the computed subdivision points need to result in the same value, or the edge may crack.

As explained in "The `precise` Qualifier" on page 56 in Chapter 2, this computation can result in different values if the expression is the same and the input values are the same but some of them are swapped due to the opposite direction of edge walking. Qualifying the results of such computations as **precise** will prevent this.

Displacement Mapping

A final technique we discuss in terms of tessellation is displacement mapping, which is merely a form of vertex texture mapping, as we described in Chapter 6, "Textures and Framebuffers." In fact, there's really not much to say, other than that you would likely use the tessellation coordinate provided to the tessellation evaluation shader in some manner to sample a texture map containing displacement information.

Adding displacement mapping to the teapot from Example 9.9 would require adding two lines to the tessellation evaluation shader, as shown in Example 9.13.

Example 9.13 Displacement Mapping in `main` Routine of the Teapot Tessellation Evaluation Shader

Click here to view code image

```
#version 420 core

layout (quads, equal_spacing, ccw) out;

uniform mat4 MV;    // Model-view matrix
uniform mat4 P;     // Projection matrix

uniform sampler2D  DisplacementMap;

void
main()
{
    vec4  p = vec4(0.0);

    float  u = gl_TessCoord.x;
    float  v = gl_TessCoord.y;

    for (int j = 0; j < 4; ++j) {
        for (int i = 0; i < 4; ++i) {
            p += B(i, u) * B(j, v) * gl_in[4*j+i].gl_Position;
        }
    }

    p += texture(DisplacementMap, gl_TessCoord.xy);

    gl_Position = P * MV * p;
}
```