Tessellation

Tessellation is the process of breaking a high-order primitive (which is known as a *patch* in OpenGL) into many smaller, simpler primitives such as triangles for rendering. OpenGL includes a fixed-function, configurable tessellation engine that is able to break up quadrilaterals, triangles, and lines into a potentially large number of smaller points, lines, or triangles that can be directly consumed by the normal rasterization hardware further down the pipeline. Logically, the tessellation phase sits directly after the vertex shading stage in the OpenGL pipeline and is made up of three parts: the tessellation control shader, the fixed-function tessellation engine, and the tessellation evaluation shader.

Tessellation Control Shaders

The first of the three tessellation phases is the tessellation control shader (TCS; sometimes known as simply the control shader). This shader takes its input from the

vertex shader and is primarily responsible for two things: the determination of the level of tessellation that will be sent to the tessellation engine, and the generation of data that will be sent to the tessellation evaluation shader that is run after tessellation has occurred.

Tessellation in OpenGL works by breaking down high-order surfaces known as *patches* into points, lines, or triangles. Each patch is formed from a number of *control points*. The number of control points per patch is configurable and set by calling **glPatchParameteri()** with `pname` set to `GL_PATCH_VERTICES` and `value` set to the number of control points that will be used to construct each patch. The prototype of **glPatchParameteri()** is

Click here to view code image

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

By default, the number of control points per patch is three. Thus, if this is what you want (as in our example application), you don't need to call it at all. The maximum number of control points that can be used to form a single patch is implementation defined, but is guaranteed to be at least 32.

When tessellation is active, the vertex shader runs once per control point, while the tessellation control shader runs in batches on groups of control points where the size of each batch is the same as the number of vertices per patch. That is, vertices are used as control points and the result of the vertex shader is passed in batches to the tessellation control shader as its input. The number of control points per patch can be changed such that the number of control points that is output by the tessellation control shader can differ from the number of control points that it consumes. The number of control points produced by the control shader is set using an output layout qualifier in the control shader's source code. Such a layout qualifier looks like this:

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

Here, `N` is the number of control points per patch. The control shader is responsible for calculating the values of the output control points and for setting the tessellation factors for the resulting patch that will be sent to the fixed-function tessellation engine. The output tessellation factors are written to the `gl_TessLevelInner` and `gl_TessLevelOuter` built-in output variables, whereas any other data that is passed down the pipeline is written to user-defined output variables (those declared using the **out** keyword, or the special built-in `gl_out` array) as normal.

Listing 3.7 shows a simple tessellation control shader. It sets the number of output control points to three (the same as the default number of input control points) using the **layout** `(vertices = 3)` **out;** layout qualifier, copies its input to its output (using

the built-in variables `gl_in` and `gl_out`), and sets the inner and outer tessellation level to 5. Higher numbers would produce a more densely tessellated output, and lower numbers would yield a more coarsely tessellated output. Setting the tessellation factor to 0 will cause the whole patch to be thrown away.

The built-in input variable `gl_InvocationID` is used as an index into the `gl_in` and `gl_out` arrays. This variable contains the zero-based index of the control point within the patch being processed by the current invocation of the tessellation control shader.

Click here to view code image

```
#version 450 core

layout (vertices = 3) out;

void main(void)
{
    // Only if I am invocation 0 ...
    if (gl_InvocationID == 0)
    {
        gl_TessLevelInner[0] = 5.0;
        gl_TessLevelOuter[0] = 5.0;
        gl_TessLevelOuter[1] = 5.0;
        gl_TessLevelOuter[2] = 5.0;
    }
    // Everybody copies their input to their output
    gl_out[gl_InvocationID].gl_Position =
        gl_in[gl_InvocationID].gl_Position;
}
```

Listing 3.7: Our first tessellation control shader

The Tessellation Engine

The tessellation engine is a fixed-function part of the OpenGL pipeline that takes high-order surfaces represented as patches and breaks them down into simpler primitives such as points, lines, or triangles. Before the tessellation engine receives a patch, the tessellation control shader processes the incoming control points and sets tessellation factors that are used to break down the patch. After the tessellation engine produces the output primitives, the vertices representing them are picked up by the tessellation evaluation shader. The tessellation engine is responsible for producing the parameters that are fed to the invocations of the tessellation evaluation shader, which it then uses to transform the resulting primitives and get them ready for rasterization.

Tessellation Evaluation Shaders

Once the fixed-function tessellation engine has run, it produces a number of output

vertices representing the primitives it has generated. These are passed to the tessellation evaluation shader. The tessellation evaluation shader (TES; also called simply the evaluation shader) runs an invocation for each vertex produced by the tessellator. When the tessellation levels are high, the tessellation evaluation shader could run an extremely large number of times. For this reason, you should be careful with complex evaluation shaders and high tessellation levels.

Listing 3.8 shows a tessellation evaluation shader that accepts input vertices produced by the tessellator as a result of running the control shader shown in Listing 3.7. At the beginning of the shader is a layout qualifier that sets the tessellation mode. In this case, we selected the mode to be triangles. Other qualifiers, `equal_spacing` and `cw`, indicate that new vertices should be generated equally spaced along the tessellated polygon edges and that a clockwise vertex winding order should be used for the generated triangles. We will cover the other possible choices in the "Tessellation" section in Chapter 8.

The remainder of the shader assigns a value to `gl_Position` just like a vertex shader does. It calculates this using the contents of two more built-in variables. The first is `gl_TessCoord`, which is the *barycentric coordinate* of the vertex generated by the tessellator. The second is the `gl_Position` member of the `gl_in[]` array of structures. This matches the `gl_out` structure written to in the tessellation control shader given in Listing 3.7. This shader essentially implements pass-through tessellation. That is, the tessellated output patch is exactly the same shape as the original, incoming triangular patch.

Click here to view code image

```
#version 450 core

layout (triangles, equal_spacing, cw) 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 3.8: Our first tessellation evaluation shader

To see the results of the tessellator, we need to tell OpenGL to draw only the outlines of the resulting triangles. To do this, we call **glPolygonMode()**, whose prototype is

Click here to view code image

```
void glPolygonMode(GLenum face,
                   GLenum mode);
```

The `face` parameter specifies which type of polygons we want to affect. Because we want to affect everything, we set it to `GL_FRONT_AND_BACK`. The other modes will be explained shortly. `mode` says how we want our polygons to be rendered. As we want to render in wireframe mode (i.e., lines), we set this to `GL_LINE`. The result of rendering our one triangle example with tessellation enabled and the two shaders of Listing 3.7 and Listing 3.8 is shown in Figure 3.1.

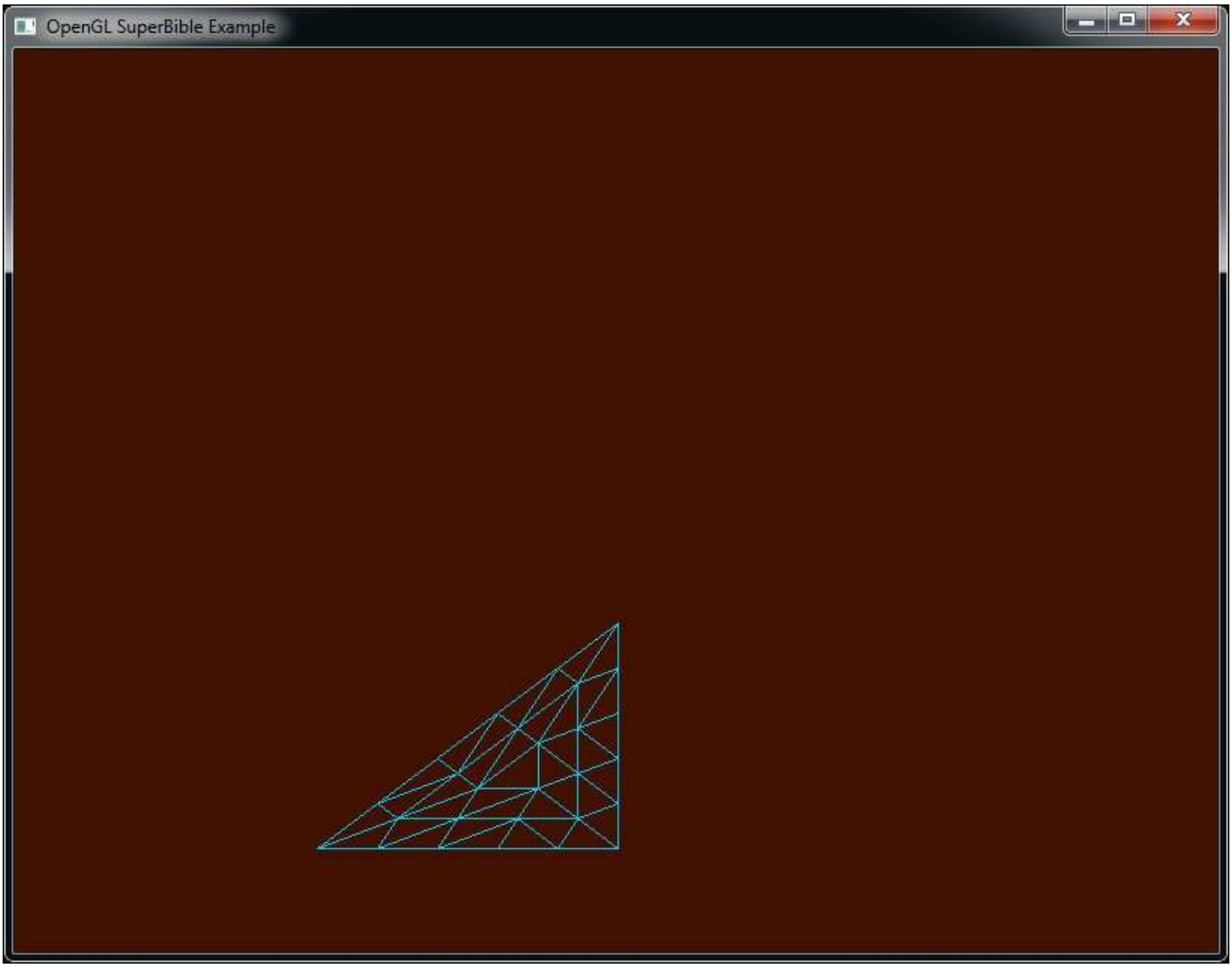

Figure 3.1: Our first tessellated triangle

Geometry Shaders

The geometry shader is logically the last shader stage in the front end, sitting after the vertex and tessellation stages and before the rasterizer. The geometry shader runs once per primitive and has access to all of the input vertex data for all of the vertices that make up the primitive being processed. The geometry shader is also unique among the shader stages in that it is able to increase or reduce the amount of data flowing through the pipeline in a programmatic way. Tessellation shaders can also increase or decrease the amount of work in the pipeline, but only implicitly by setting the tessellation level

for the patch. Geometry shaders, in contrast, include two functions—`EmitVertex()` and `EndPrimitive()`—that explicitly produce vertices that are sent to primitive assembly and rasterization.

Another unique feature of geometry shaders is that they can change the primitive mode mid-pipeline. For example, they can take triangles as input and produce a bunch of points or lines as output, or even create triangles from independent points. An example geometry shader is shown in Listing 3.9.

Click here to view code image

```
#version 450 core

layout (triangles) in;
layout (points, 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();
    }
}
```

Listing 3.9: Our first geometry shader

The shader shown in Listing 3.9 acts as another simple pass-through shader that converts triangles into points so that we can see their vertices. The first layout qualifier indicates that the geometry shader is expecting to see triangles as its input. The second layout qualifier tells OpenGL that the geometry shader will produce points and that the maximum number of points that each shader will produce will be three. In the `main` function, a loop runs through all of the members of the `gl_in` array, which is determined by calling its `.length()` function.

We actually know that the length of the array will be three because we are processing triangles and every triangle has three vertices. The outputs of the geometry shader are again similar to those of a vertex shader. In particular, we write to `gl_Position` to set the position of the resulting vertex. Next, we call `EmitVertex()`, which produces a vertex at the output of the geometry shader. Geometry shaders automatically call `EndPrimitive()` at the end of your shader, so calling this function explicitly is not necessary in this example. As a result of running this shader, three vertices will be produced and rendered as points.

By inserting this geometry shader into our simple one tessellated triangle example, we

obtain the output shown in Figure 3.2. To create this image, we set the point size to 5.0 by calling **glPointSize()**. This makes the points large and highly visible.

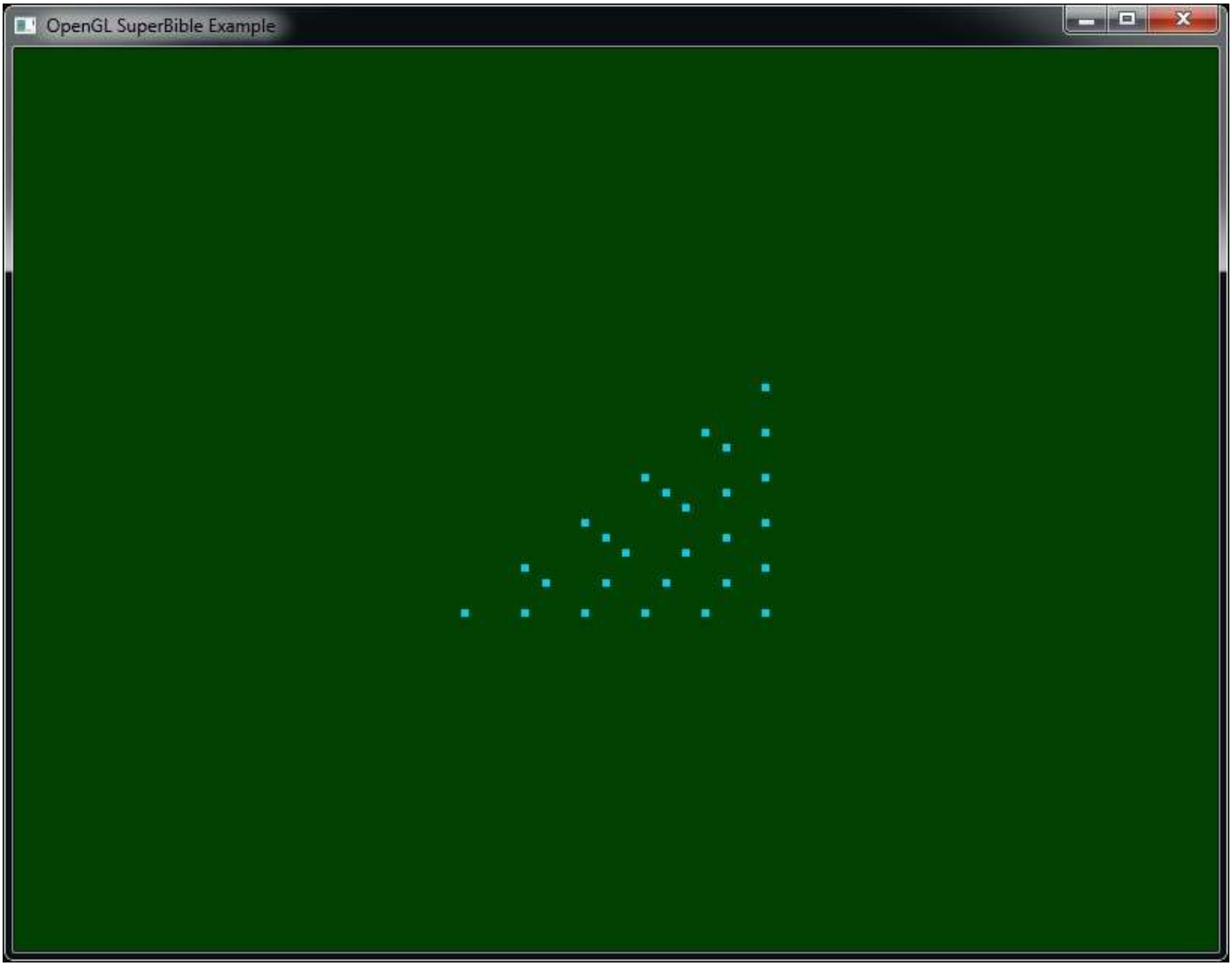

Figure 3.2: Tessellated triangle after adding a geometry shader

Primitive Assembly, Clipping, and Rasterization

After the front end of the pipeline has run (which includes vertex shading, tessellation, and geometry shading), a fixed-function part of the pipeline performs a series of tasks that take the vertex representation of our scene and convert it into a series of pixels, which in turn need to be colored and written to the screen. The first step in this process is primitive assembly, which is the grouping of vertices into lines and triangles. Primitive assembly still occurs for points, but it is trivial in that case.

Once primitives have been constructed from their individual vertices, they are *clipped* against the displayable region, which usually means the window or screen, but can also be a smaller area known as the *viewport*. Finally, the parts of the primitive that are determined to be potentially visible are sent to a fixed-function subsystem called the

rasterizer. This block determines which pixels are covered by the primitive (point, line, or triangle) and sends the list of pixels on to the next stage—that is, fragment shading.

Clipping

As vertices exit the front end of the pipeline, their position is said to be in *clip space*. This is one of the many coordinate systems that can be used to represent positions. You may have noticed that the `gl_Position` variable that we have written to in our vertex, tessellation, and geometry shaders has a **vec4** type, and that the positions we have produced by writing to it are all four-component vectors. This is what is known as a *homogeneous* coordinate. The homogeneous coordinate system is used in projective geometry because much of the math ends up being simpler in homogeneous coordinate space than it does in regular Cartesian space. Homogeneous coordinates have one more component than their equivalent Cartesian coordinate, which is why our three-dimensional position vector is represented as a four-component variable.

Although the output of the front end is a four-component homogeneous coordinate, clipping occurs in Cartesian space. Thus, to convert from homogeneous coordinates to Cartesian coordinates, OpenGL performs a *perspective division*, which involves dividing all four components of the position by the last, *w* component. This has the effect of projecting the vertex from the homogeneous space to the Cartesian space, leaving *w* as 1.0. In all of the examples so far, we have set the *w* component of `gl_Position` as 1.0, so this division has not had any effect. When we explore projective geometry in a short while, we will discuss the effect of setting *w* to values other than 1.0.

After the projective division, the resulting position is in *normalized device space*. In OpenGL, the visible region of normalized device space is the volume that extends from −1.0 to 1.0 in the *x* and *y* dimensions and from 0.0 to 1.0 in the *z* dimension. Any geometry that is contained in this region may become visible to the user and anything outside of it should be discarded. The six sides of this volume are formed by planes in three-dimensional space. As a plane divides a coordinate space in two, the volumes on each side of the plane are called *half-spaces*.

Before passing primitives on to the next stage, OpenGL performs clipping by determining which side of each of these planes the vertices of each primitive lie on. Each plane effectively has an "outside" and an "inside." If a primitive's vertices all lie on the "outside" of any one plane, then the whole thing is thrown away. If all of primitive's vertices are on the "inside" of all the planes (and therefore inside the view volume), then it is passed through unaltered. Primitives that are partially visible (which means that they cross one of the planes) must be handled specially. More details about how this works is given in the "Clipping" section in Chapter 7.

Viewport Transformation

After clipping, all of the vertices of the geometry have coordinates that lie between −1.0 and 1.0 in the x and y dimensions. Along with a z coordinate that lies between 0.0 and 1.0, these are known as normalized device coordinates. However, the window that you're drawing to has coordinates that usually[1] start from (0, 0) at the bottom left and range to $(w-1, h-1)$, where w and h are the width and height of the window in pixels, respectively. To place your geometry into the window, OpenGL applies the *viewport transform*, which applies a scale and offset to the vertices' normalized device coordinates to move them into *window coordinates*. The scale and bias to apply are determined by the viewport bounds, which you can set by calling **glViewport()** and **glDepthRange()**. Their prototypes are

1. It's possible to change the coordinate convention such that the (0, 0) origin is at the upper-left corner of the window, which matches the convention used in some other graphics systems.

Click here to view code image

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

and

Click here to view code image

```
void glDepthRange(GLdouble nearVal, GLdouble farVal);
```

This transform takes the following form:

$$\begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{p_x}{2} x_d + o_x \\ \frac{p_y}{2} y_d + o_y \\ \frac{f-n}{2} z_d + \frac{n+f}{2} \end{pmatrix}$$

Here, x_w, y_w, and z_w are the resulting coordinates of the vertex in window space, and x_d, y_d, and z_d are the incoming coordinates of the vertex in normalized device space. p_x and p_y are the width and height of the viewport in pixels, and n and f are the near and far plane distances in the z coordinate, respectively. Finally, o_x, o_y, and o_z are the origins of the viewport.

Culling

Before a triangle is processed further, it may be optionally passed through a stage called *culling*, which determines whether the triangle faces toward or away from the viewer and can decide whether to actually go ahead and draw it based on the result of this computation. If the triangle faces toward the viewer, then it is considered to be *front-facing*; otherwise, it is said to be *back-facing*. It is very common to discard triangles that are back-facing because when an object is closed, any back-facing triangle will be hidden by another front-facing triangle.

To determine whether a triangle is front- or back-facing, OpenGL will determine its

signed area in window space. One way to determine the area of a triangle is to take the cross product of two of its edges. The equation for this is

$$a = \frac{1}{2}\sum_{i=0}^{n-1} x_w^i y_w^{i\oplus 1} - x_w^{i\oplus 1} y_w^i$$

Here, x_w^i and y_w^i are the coordinates of the *i*th vertex of the triangle in window space and $i \oplus 1$ is $(i+1)$ mod 3. If the area is positive, then the triangle is considered to be front-facing; if it is negative, then it is considered to be back-facing. The sense of this computation can be reversed by calling **glFrontFace()** with `dir` set to either `GL_CW` or `GL_CCW` (where `CW` and `CCW` stand for clockwise and counterclockwise, respectively). This is known as the *winding order* of the triangle, and the clockwise or counterclockwise terms refer to the order in which the vertices appear in window space. By default, this state is set to `GL_CCW`, indicating that triangles whose vertices are in counterclockwise order are considered to be front-facing and those whose vertices are in clockwise order are considered to be back-facing. If the state is `GL_CW`, then *a* is simply negated before being used in the culling process. Figure 3.3 shows this pictorially for the purpose of illustration.

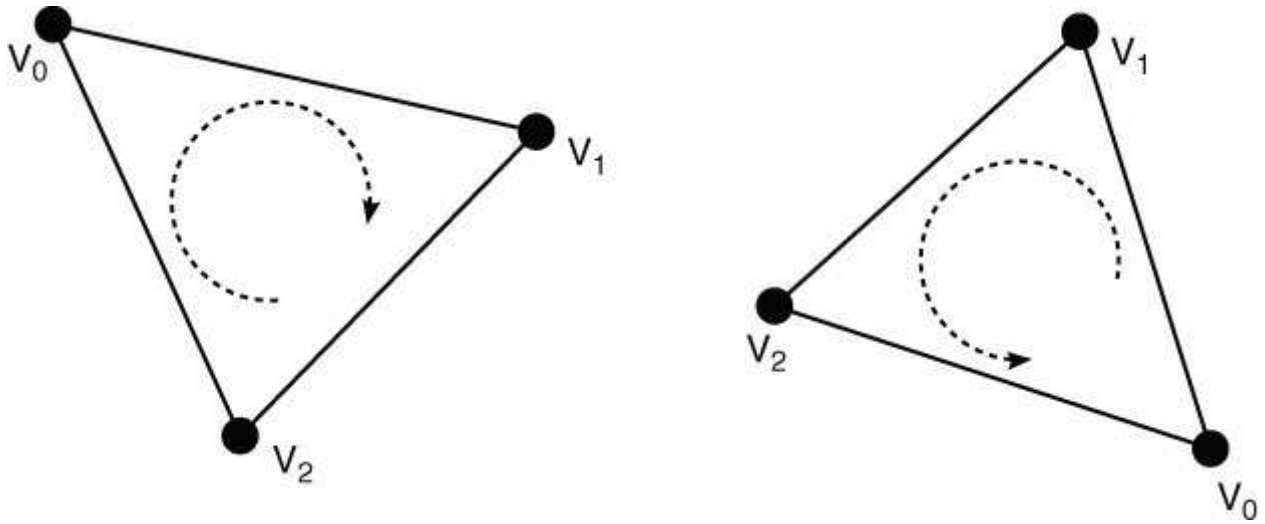

Figure 3.3: Clockwise (left) and counterclockwise (right) winding order

Once the direction that the triangle is facing has been determined, OpenGL is capable of discarding either front-facing, back-facing, or even both types of triangles. By default, OpenGL will render all triangles, regardless of which way they face. To turn on culling, call **glEnable()** with `cap` set to `GL_CULL_FACE`. When you enable culling, OpenGL will cull back-facing triangles by default. To change which types of triangles are culled, call **glCullFace()** with `face` set to `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`.

As points and lines don't have any geometric area,[2] this facing calculation doesn't apply to them and they can't be culled at this stage.

2. Obviously, once they are rendered to the screen, points and lines have area; otherwise, we wouldn't be able to see them. However, this area is artificial and can't be calculated directly from their vertices.

Rasterization

Rasterization is the process of determining which fragments might be covered by a primitive such as a line or a triangle. There are myriad algorithms for doing this, but most OpenGL systems will settle on a half-space–based method for triangles, as it lends itself well to parallel implementation. Essentially, OpenGL will determine a bounding box for the triangle in window coordinates and test every fragment inside it to determine whether it is inside or outside the triangle. To do this, it treats each of the triangle's three edges as a half-space that divides the window in two.

Fragments that lie on the interior of all three edges are considered to be inside the triangle and fragments that lie on the exterior of any of the three edges are considered to be outside the triangle. Because the algorithm to determine which side of a line a point lies on is relatively simple and is independent of anything besides the position of the line's endpoints and of the point being tested, many tests can be performed concurrently, providing the opportunity for massive parallelism.