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. In fact, we can actually configure up to four output *streams* and use the geometry shader to send its output to whichever one it chooses. This approach can be used, for example, to sort geometry or to render some primitives while storing other geometry in transform feedback buffers. Nevertheless, several major limitations apply 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 store data into transform feedback buffers, just the first stream may be rendered—the others are used for storage only. If your application fits within these constraints, however, this functionality 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. Like most other output layout qualifiers, the `stream` qualfier 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:

Click here to view code image

```
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 // All of '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 which to send the output:

Click here to view code image

```
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 truly useful only when you're rendering with a geometry shader active. When the new adjacency primitive types are used, for each line or triangle passed into the geometry shader, the shader has access not only to the vertices defining that primitive, but also 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.

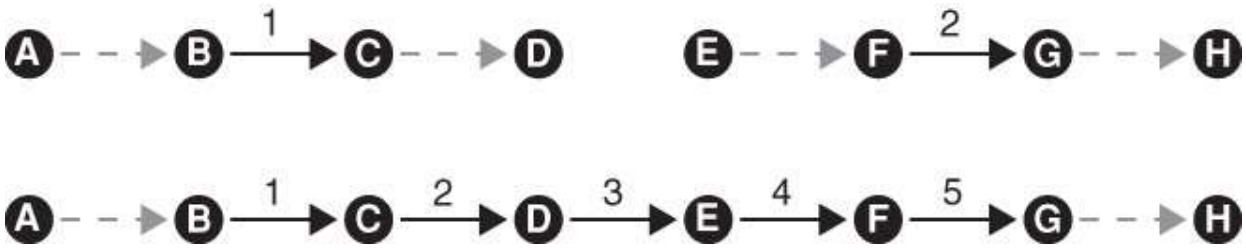

Figure 8.20: Lines produced using lines with adjacency primitives

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 vertices. In turn, the inputs to the geometry shader are six-element arrays.

As before, because 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 case.

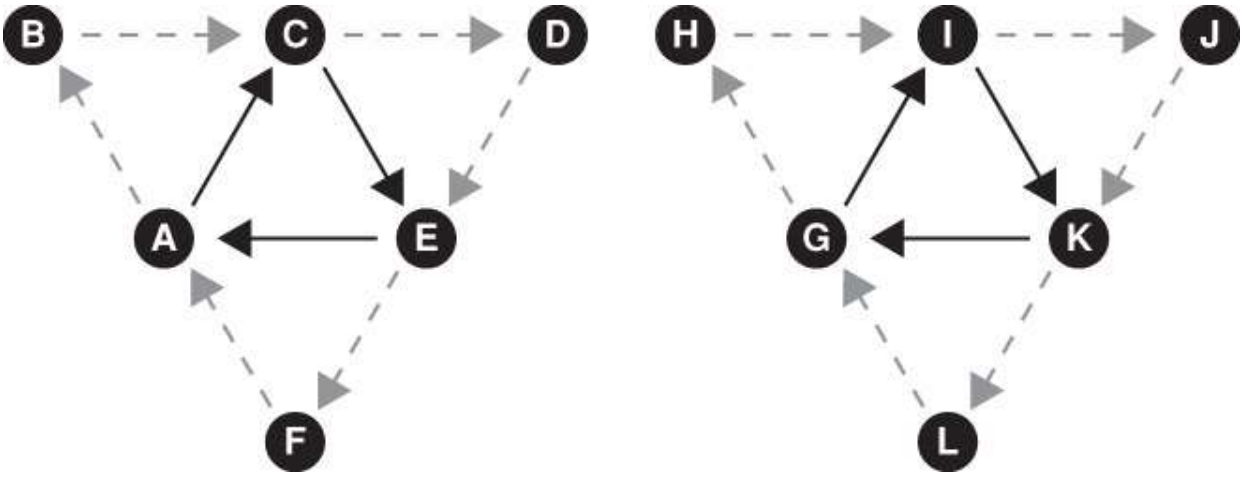

Figure 8.21: Triangles produced using `GL_TRIANGLES_ADJACENCY`

The final, perhaps most complex (or, alternatively, most difficult to understand) of these primitive types is `GL_TRIANGLE_STRIP_ADJACENCY`. This primitive represents a triangle strip in which every other vertex (the first, third, fifth, seventh, ninth, and so on) forms 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.

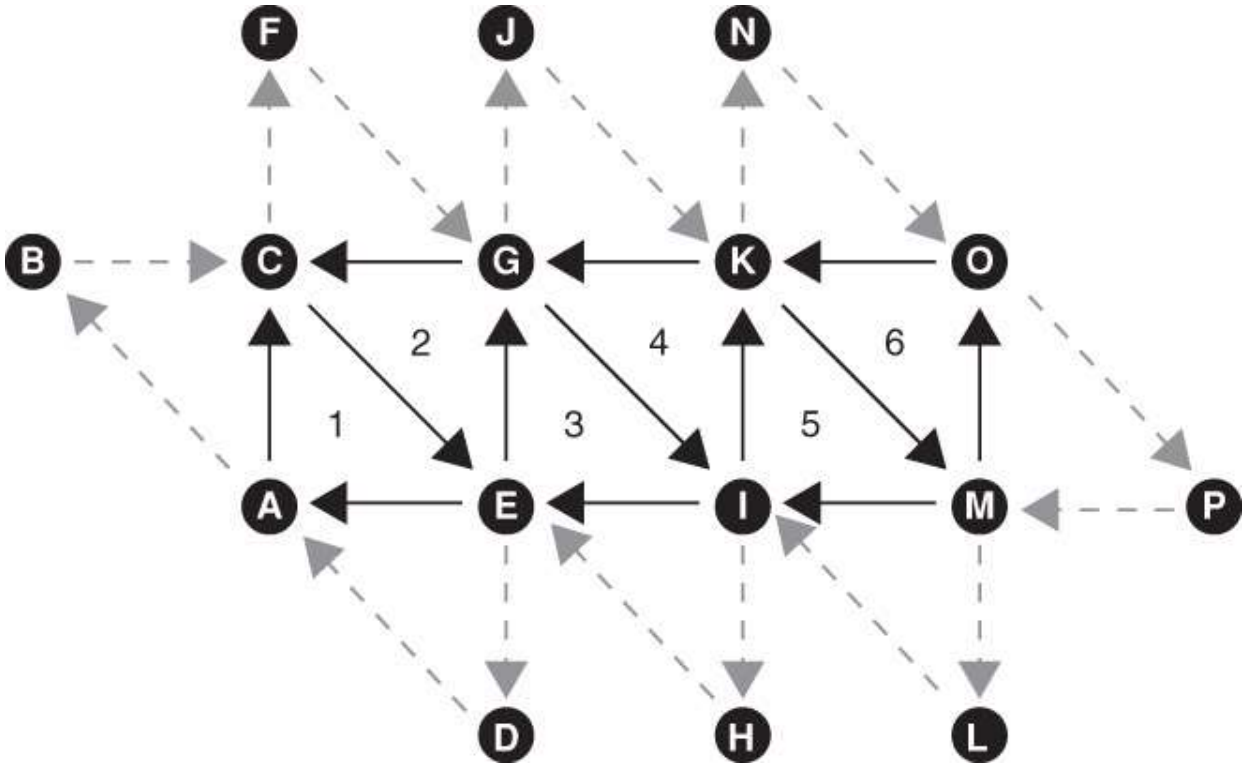

Figure 8.22: Triangles produced using `GL_TRIANGLE_STRIP_ADJACENCY`

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

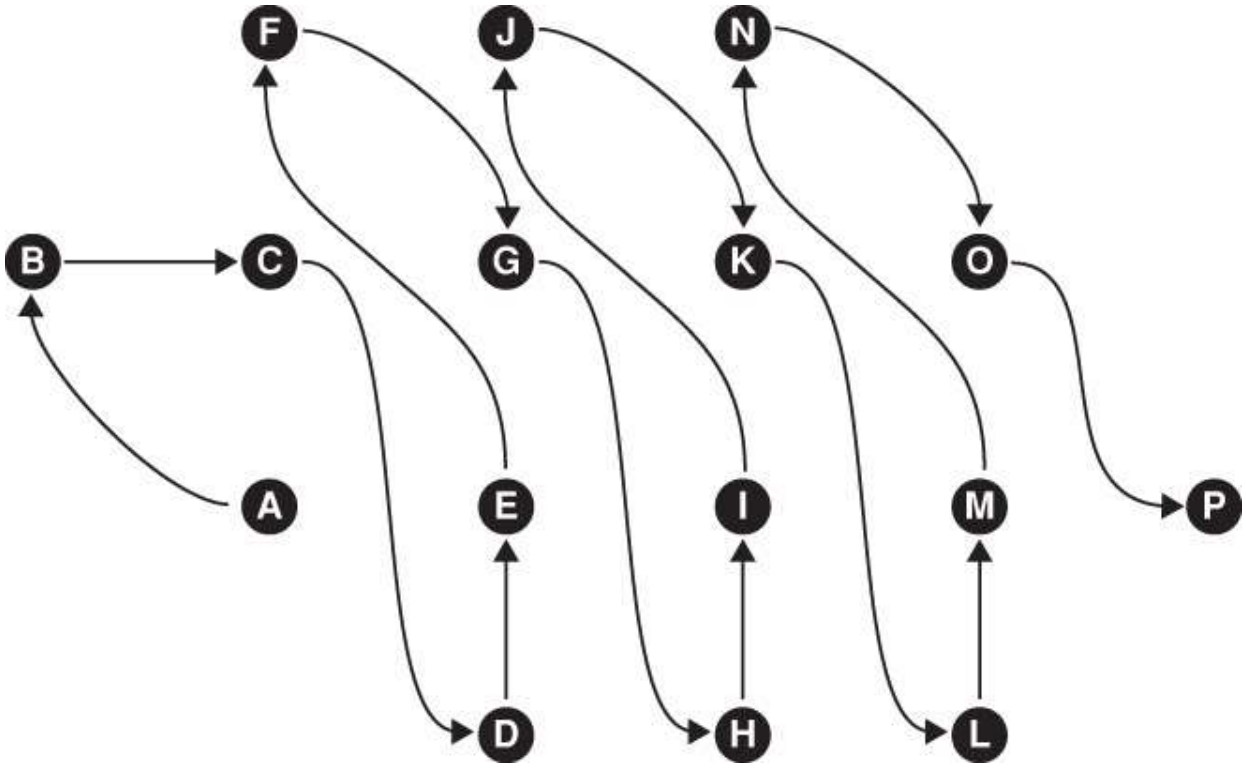

Figure 8.23: Ordering of vertices for `GL_TRIANGLE_STRIP_ADJACENCY`

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.

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 the 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 is 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.

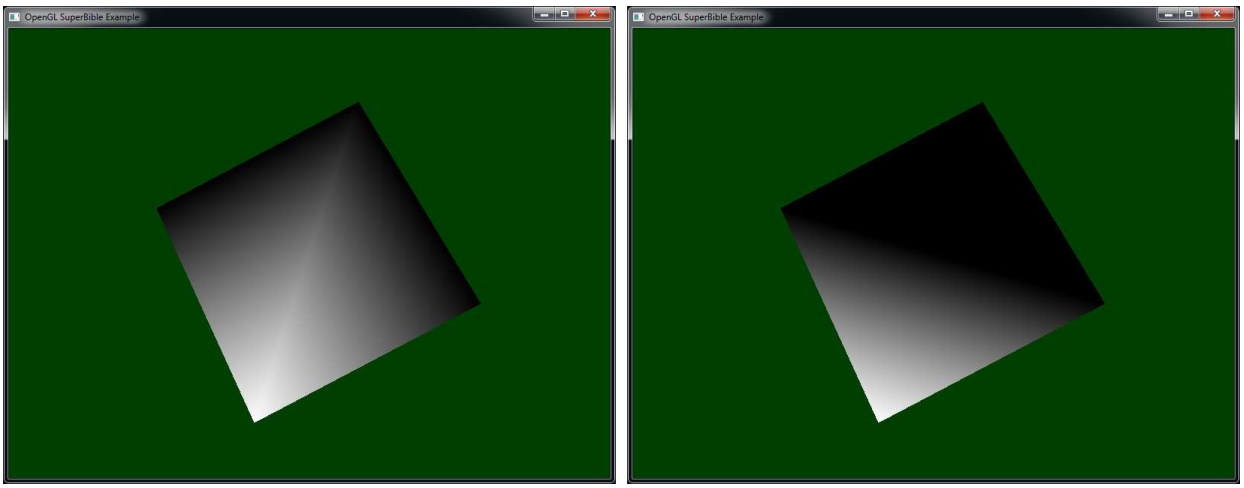

Figure 8.24: Rendering a quad using a pair of triangles

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 bottom-most 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 bottom-most triangle, which contains one white vertex and two black ones, therefore displaying a black to white gradient.

The reason for these differences 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; as a consequence, 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, even 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 problem? Well, the geometry shader is able to accept primitives with the `GL_LINES_ADJACENCY` type, and each of these primitives has four vertices—exactly enough to represent a quad. Thus, 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 that the output of the geometry shader can be only points, lines, or triangles. Thus 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 earlier. 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.

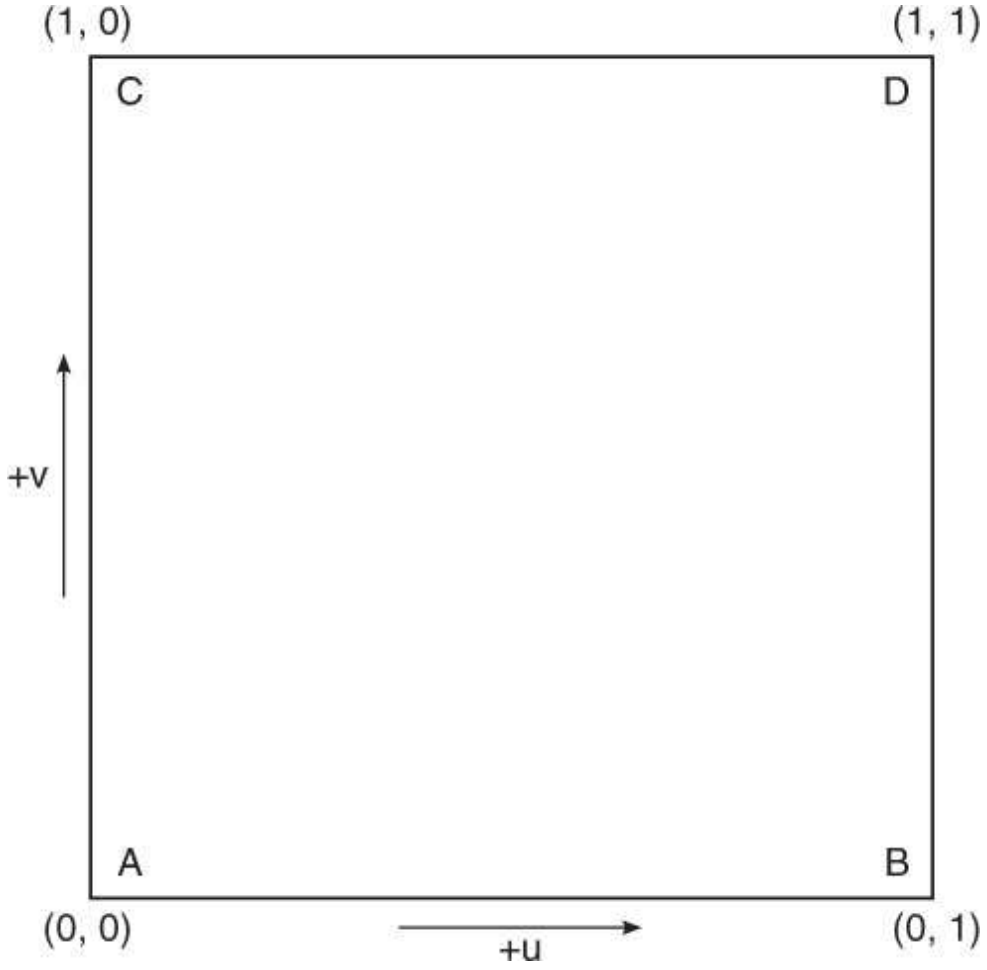

Figure 8.25: Parameterization of a quad

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 vertices *A* and *B* and between vertices *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 the 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 for each attribute at any point inside the quad.

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 in Figure 8.24 is shown in Figure 8.26.

Click here to view code image

```
#version 450 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 it's a flat attribute,
    // 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();
}
```

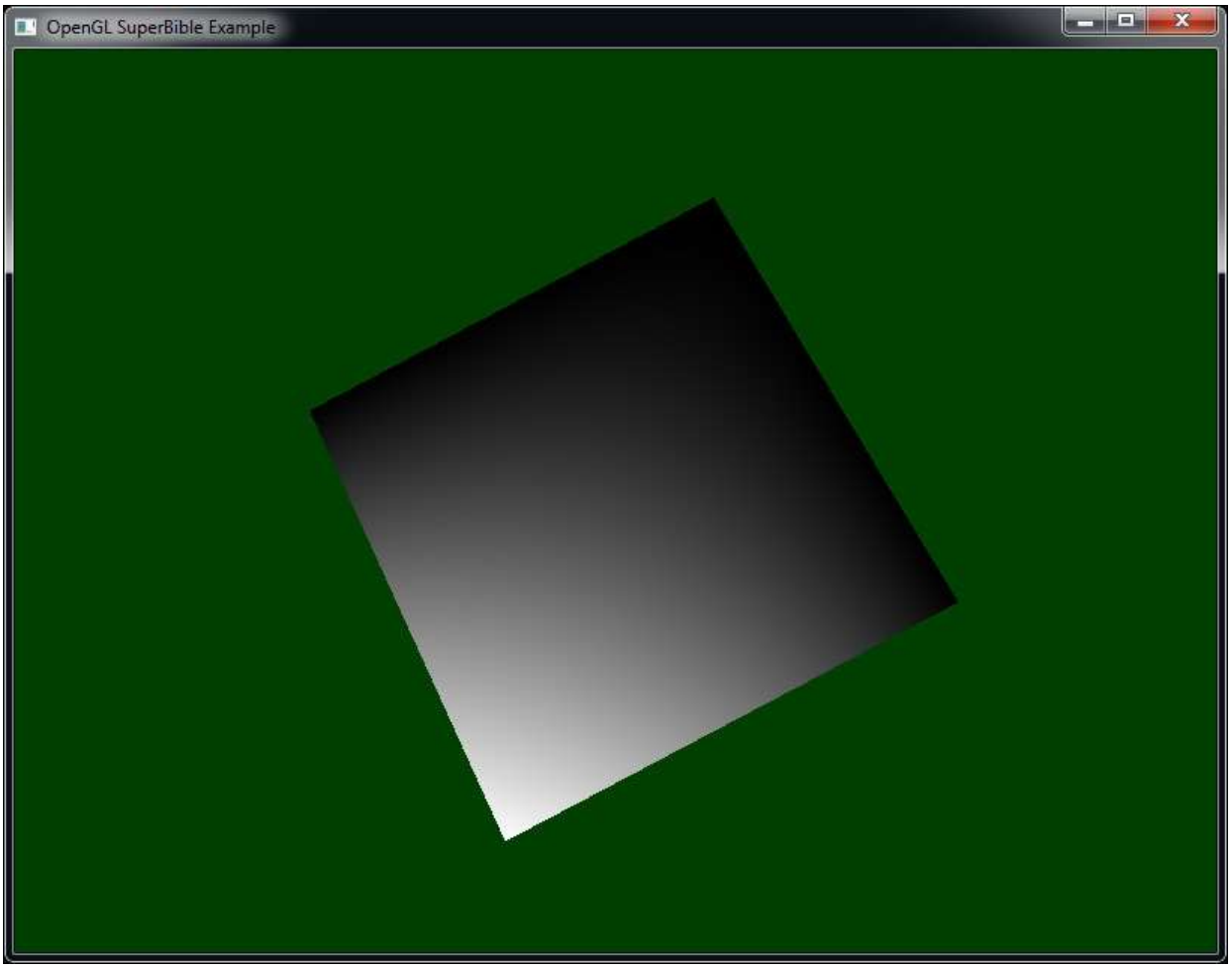

Figure 8.26: Quad rendered using a geometry shader

Listing 8.34: Geometry shader for rendering quads

Click here to view code image

```
#version 450 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

In the "Viewport Transformation" section in Chapter 3, you learned about the viewport transformation and discovered 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 taking over the whole display. However, it's possible to move the viewport around and draw into multiple virtual windows within a single larger framebuffer. Furthermore, OpenGL allows you to use multiple viewports *at the same time*—a feature known as viewport arrays.

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

Click here to view code image

```
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. 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, so `index` can range from 0 to 15. Likewise, each viewport has its own depth range, which can be specified by calling **`glDepthRangeIndexed()`**, whose prototype is

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

Click here to view code image

```
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

Click here to view code image

```
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.

Click here to view code image

```
#version 450 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 in 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 that it retrieves from the uniform block, `transform_block`. Of course, a more complex shader could be constructed, but this example is sufficient to demonstrate the direction of transformed geometry into a number of different viewports. We have implemented this code in the `multipleviewport` example, and the result of running this shader on our simple spinning cube is shown in Figure 8.27.

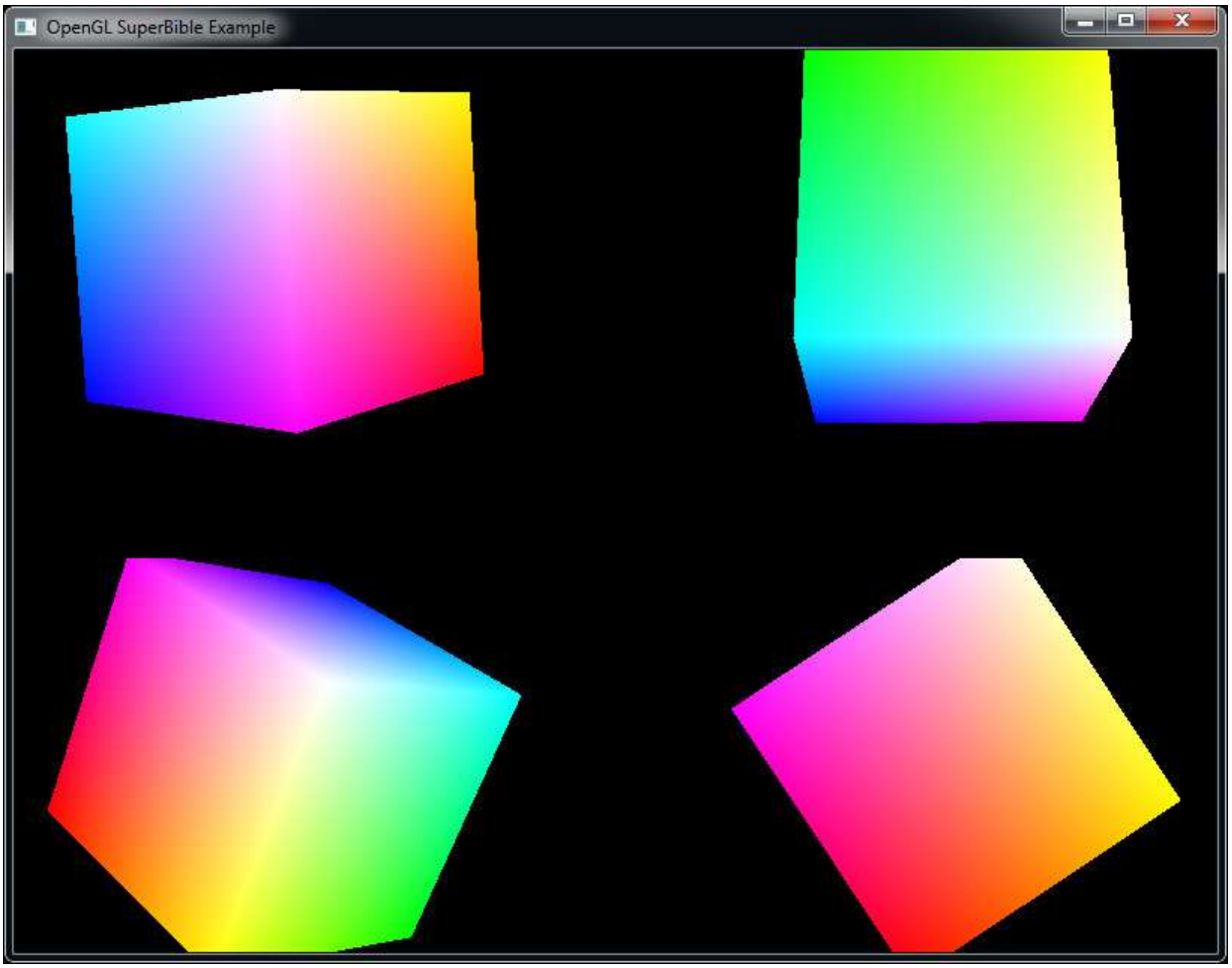

Figure 8.27: Result of rendering to multiple viewports

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. Where the cubes extend past the edges of their respective viewports, their corners are cut off by OpenGL's clipping stage.