Tutorial4: Using Indices and Geometry Shaders (C /SDL)

Contents

C/SDL Tutorial Series

Overview

This tutorial is designed to help explain how to use indices and geometry shaders in the OpenGL 3.2 core profile.

In tutorial3 we created a tetrahedron using 12 vertices, 3 for each triangle. In this tutorial we will use only 4 vertices to make the same tetrahedron using indices. We will also make use of geometry shaders, which are now a standard part of the OpenGL 3.2 specification, to create a second tetrahedron intersecting our original tetrahedron.

Changes to drawing the scene

Indices are an easy way to refer to vertex that can be used to create primitives. The indices we use will describe a triangle strip. A triangle strip is useful if you are connecting multiple triangles together as it reuses vertex data. Let's take a look at the code used to define the tetrahedron, colors, and indices.

```
/* The four vericies of a tetrahedron */
const GLfloat tetrahedron[4][3] = {
{  1.0,  1.0,  1.0  },   /* index 0 */
{ -1.0, -1.0,  1.0  },   /* index 1 */
{ -1.0,  1.0, -1.0  },   /* index 2 */
{  1.0, -1.0, -1.0  } }; /* index 3 */

/* Color information for each vertex */
const GLfloat colors[4][3] = {
{  1.0,  0.0,  0.0  },   /* red */
{  0.0,  1.0,  0.0  },   /* green */
{  0.0,  0.0,  1.0  },   /* blue */
{  1.0,  1.0,  1.0  } }; /* white */

const GLubyte tetraindicies[6] = { 0, 1, 2, 3, 0, 1 };
```

As you can see tetrahedron[][] defines the four vertices needed to create a tetrahedron. The tetraindices const defines how to connect these vertices together. Since we are using a triangle strip, we only need 6 indices to create 4 triangles. Triangle strips use 3 indices to connect the first triangle and create another triangle for each additional index defined using the last two indices used. For instance this will create triangles using indices 0,1,2 then 1,2,3 then 2,3,0 and finally 3,0,1. To use the indices we have defined, we need to bind them to a VBO. The following code is used for this process:

```
    /* Bind our third VBO as being the active buffer and storing vertex array indicies */
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vbo[2]);

    /* Copy the index data from tetraindicies to our buffer
     * 6 * sizeof(GLubyte) is the size of the index array, since it contains 6 GLbyte values */
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(GLubyte), tetraindicies, GL_STATIC_DRAW);
```

Note that unlike our coordinate and color data, we do not bind our index VBO to an attribute location for use in our shader programs. Using indices also means using one of the DrawElements functions instead of DrawArrays.

```
        /* Invoke glDrawElements telling it to draw a triangle strip using 6 indicies */
        glDrawElements(GL_TRIANGLE_STRIP, 6, GL_UNSIGNED_BYTE, 0);
```

This tutorial also makes use of geometry shaders and so we must perform the same operations of creating a shader object, associating source code, compiling, and attaching as we do for vertex and fragment shaders.

```
    /* Read our shaders into the appropriate buffers */
    geometrysource = filetobuf("tutorial4.geom");

    /* Assign our handles a "name" to new shader objects */
    geometryshader = glCreateShader(GL_GEOMETRY_SHADER);

    /* Compile our shader objects */
    glCompileShader(geometryshader);

    /* Attach our shaders to our program */
    glAttachShader(shaderprogram, geometryshader);
```

Changes in the vertex shader

In this tutorial we will be shifting the mvpmatrix multiplication part from the vertex shader to the geometry shader. This is done for simplicity as the intersection operation we perform on the tetrahedron model is easiest to do on a normalized object (Before transformations).

The geometry shader

The geometry shader receives data in the form of primatives, in our case triangles, from the vertex shader. This means that all incoming data types are an array of the number of vertices of our primitive, in our case 3 for a triangle. The geometry shader should specify the type of primitive output by our shader as well as the maximum number of vertices the program should generate. Optionally we can even declare the type of input primitive the shader expects to be receiving.

```
// Declare what type of incoming primitive our geometry shader is receiving
layout(triangles) in;

// Declare what type of primitives we are creating and the maximum amount of vertices we will output per use of the geometry shader.
// We will be outputting 6 vertices per use of this shader, creating 2 triangles.
layout(triangle_strip, max_vertices = 6) out;
```

In GLSL 1.50 geometry shader input positions come from an array called gl_in. For example: gl_in[0].gl_Position holds the value we assigned gl_Position for the first vertex of our primitive during the vertex shader stage of our shading program. The geometry shader is responsible for taking these vertex positions and creating primitives. We take our input positions, perform any calculations we want on them then use the EmitVertex() function to create a vertex using values assigned to gl_Position and any output parameters we specify, such as color. After we have used EmitVertex enough times to create a primitive, we call EndPrimitive().

Full source code

Full source code is available in a zip file here (Note: broken link removed).

Compilation

On linux:

gcc utils.c tutorial4.c -o tutorial4 -lGL $(sdl-config --cflags --libs)

If you have SDL-1.2 and SDL-1.3 both installed, be sure to run the 1.3 version of sdl-config. For example if you installed SDL-1.3 in /usr/local:

gcc utils.c tutorial4.c -o tutorial4 -lGL $(/usr/local/bin/sdl-config --cflags --libs)

Execution

./tutorial4

The result should be a 512x512 window centered on your display showing two intersected rotating tetrahedrons.

Notes

You may notice that the colors are blended between each vertex and not solid like in tutorial3. This is because the default color blending method is smooth. However if you wish to change the blending to be the same as in tutorial 3, we have to declare our color parameter as being flat. To do this open the shader files and change each declaration of the color to include the prefix flat. For example:

```
In tutorial4.vert:
flat out vec3 geom_Color;

In tutorial4.geom:
flat in vec3 geom_Color[3];
flat out vec3 ex_Color;

In tutorial4.frag:
flat in vec3 ex_Color;
flat out vec4 gl_FragColor;
```