CS3388B: Lecture 16

March 28, 2023

16 Geometry and Tesselation Shaders: Part II

16.1 Tesselation Control Shaders

A TCS is invoked on a single patch of vertices.

- The number of vertices N in a patch is specified by:

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

- **The TCS is invoked once per vertex per patch.** Which vertex a particular invocation corresponds to is held in the `gl_InvocationID` built-in input variables.

- All inputs from vertex shaders to the TCS are aggregated into arrays, based on the size of the input patch N. User-defined outputs from the vertex shader can be accessed like:

```
in vec3 color_vs[];
in float someFloat_vs[];
```

- The built-in outputs from the vertex shader can be accessed like:

```
gl_in[gl_InvocationID].gl_Position;
```

- The TCS should write user-defined per-vertex data out to an array using `gl_InvocationID` as index:

```
out vec3 color_tcs[];
//...
color_tcs[gl_InvocationID] = color_vs[gl_InvocationID];
```

- The TCS should write per-vertex data out to an array using `gl_InvocationID` as index:

```
out vec3 color_tcs[];
//...
color_tcs[gl_InvocationID] = color_vs[gl_InvocationID];
```

- The TCS must write to `gl_TessLevelOuter[]` and `gl_TessLevelInner[]` to contol the tesselation levels. The number of indices to use for each outer and inner depends on the

abstract patch type defined in the Tesselation evaluation shader.

  - For triangles: 3 outer levels, 1 inner level;
  - For quads: 4 outer levels, 2 inner levels;
  - For isolines: 2 outer levels, 0 inner levels;

- The TCS must write to `gl_out[].gl_Position`.

```
//A Tesselation Control Shader
#version 400

layout (vertices = 3) out;

in vec3 color_vs[];
out vec3 color_tcs[];

uniform float outerTess;
uniform float innerTess;

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

    color_tcs[gl_InvocationID] = color_vs[gl_InvocationID];

    gl_TessLevelOuter[0] = outerTess;
    gl_TessLevelOuter[1] = outerTess;
    gl_TessLevelOuter[2] = outerTess;
    gl_TessLevelInner[0] = innerTess;
}
```

Note: `gl_InvocationID` is the index within the patch of the incoming vertex. If your vertex buffer has:

```
-1, 1, 0,
1, 1, 0,
0, 0, 0
```

Then vertex (-1, 1, 0) has `gl_InvocationID` 0 and `gl_in[0]` corresponds to whatever the vertex shader outputs when processing the vertex (-1,1,0).

16.2 Tesselation Evaluation Shader

Depending on the tesselation levels specified by the tesselation control shader, the fixed-function tesselation primitive generation produces extra vertices from the input patch vertices and the abstract primitives they define. They type of abstract primitive is defined in the Tesselation Evaluation Shader (TES).

Moreover, the vertices created during tesselation must have vertex attributes assigned to them. This is the job of the TES. It acts a secondary vertex shader for the tesselation-created vertices.

Vertex attributes output from the vertex shader are not automatically interpolated during tesselation primitive generation.

The TES must itself compute the correct vertex attributes, including position, for each of the newly generated vertices. The TES is invoked once per new vertex. It must compute the:

1. Vertex position (in clip space or whatever is expected by the geometry shader).
2. Any user-defined vertex attributes expected by the geometry shader or fragment shader.

In particular, the TES has the following specifications:

1. The abstract patch type and *how* tesselation should be applied is defined by the layout parameter:

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

   The abstract patch types are triangles, quads, and isolines. The spacing types are `equal_spacing`, `fractional_even_spacing`, `fractional_odd_spacing`.

2. All inputs from TCS to the TES are aggregated into arrays, based on the number of vertices in the abstract patch type.

   User-defined inputs and built-in inputs are accessed just as the TES, but using the vertex indices of the abstract patch rather than `gl_InvocationID`.

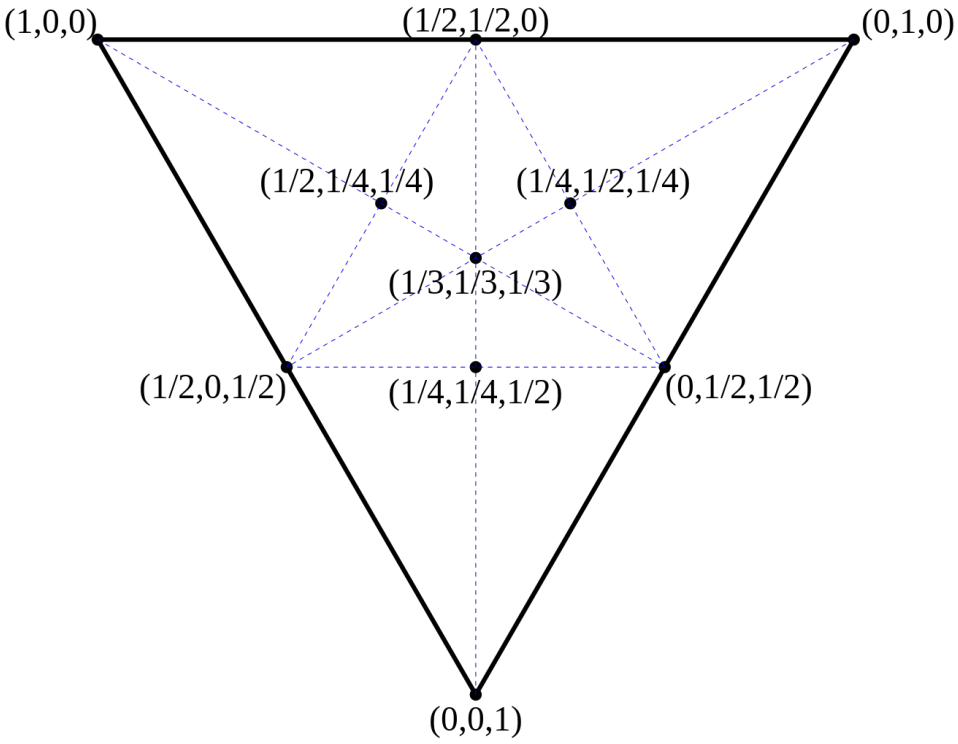
3. All outputs of the TES are per-vertex and are thus **not** arrays. This includes the normal vertex shader outputs like `gl_Position`, and any other vertex attribute expected by the geometry/fragment shader.

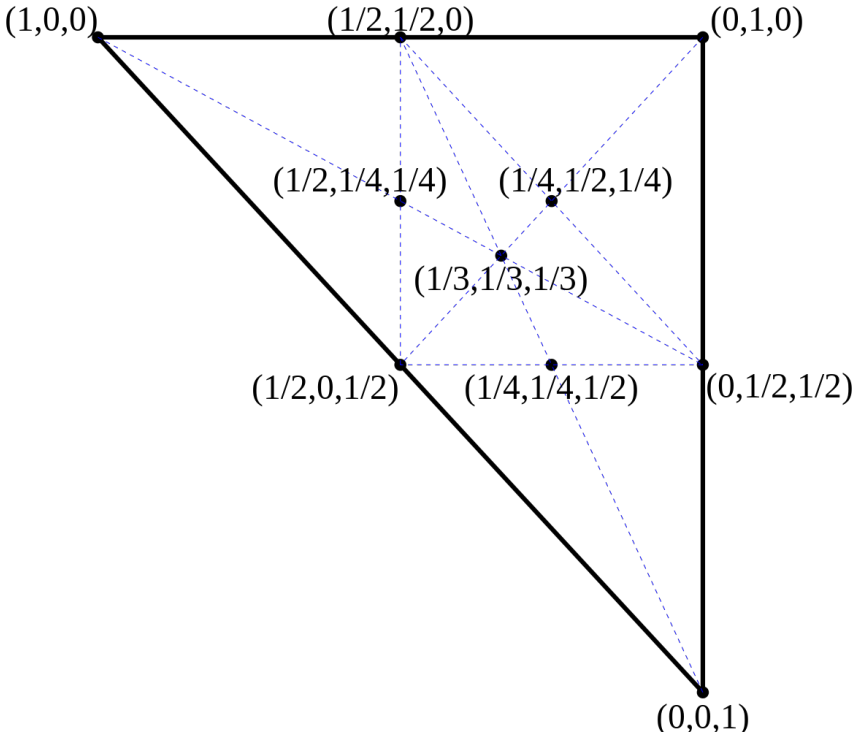
4. When the abstract patch is a quad, the outputs of the TES are computed **similar to texture coordinates**, using a normalized 2D point to interpolate between the 4 corners of the quad. It uses the first two indices from `vec3 gl_TessCoord`.

5. When the abstract patch is a triangle, the outputs of the TES are computed using **barycentric coordinates** and three indices from `vec3 gl_TessCoord`.

16.2.1 Barycentric Coordinates

Without going into the math of it, **barycentric coordinates** give the position within a triangle as a 3D vector. The vector gives the **weights in a weighted sum** of the three corners.





```
//A Tesselation Evaluation Shader
#version 400

layout (triangles, equal_spacing) in;

uniform mat4 MVP;

in vec3 color_tcs[];
out vec3 color_gs;

void main() {

    vec4 p1= gl_in[0].gl_Position;
    vec4 p2= gl_in[1].gl_Position;
    vec4 p3= gl_in[2].gl_Position;

    gl_Position = vec4(0.0);
    gl_Position += gl_TessCoord.x *p1;
    gl_Position += gl_TessCoord.y *p2;
    gl_Position += gl_TessCoord.z *p3;

    color_gs = vec3(0.0);
    color_gs += gl_TessCoord.x *color_tcs[0];
    color_gs += gl_TessCoord.y *color_tcs[1];
    color_gs += gl_TessCoord.z *color_tcs[2];
}
```

When using a Tesselation evaluation shader the outputs must match the inputs of the geometry/fragment shader which is next in the render pipeline.

16.3 Geometry Shaders with Tesselation Shaders: A Use Case

One possibly annoying part of tesselation shaders is their implicit interpolation of vertex attributes. Since each vertex attribute output from a TES is computed using `gl_TessCoords`, the vertices output from tesselation may "lose" the fact they are now independent primitives generated from pre-existing ones.

Consider defining a vertex attribute for each corner of a tesselation-generated triangle as:

```
vec3 attrib1(1, 0, 0);
vec3 attrib2(0, 1, 0);
vec3 attrib3(0, 0, 1);
```

Since each new vertex is computing using `gl_TessCoord`, it is impossible (or at least very hard) to determine which vertex corresponds to which discrete primitive.

But, this is exactly the job of the geometry shader! To process per-primitive attributes. Once a tesselation evaluation shader has created vertices and vertex attributes, those go into the geometry shader as if they are full-fledged primitives. The geometry shader does not know if they are user-defined vertices or tesselation-generated vertices.

So long as the input primitive type of the geometry shader matches the abstract patch type of a Tesselation Evaluation Shader, you are in promising territory.

```
//A Geometry Shader
#version 400

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

in vec3 color_tes[];
flat out vec3 color_gs;
out vec3 triDistance;

void main() {
    gl_Position = gl_in[0].gl_Position;
    color_gs = color_tes[0];
    triDistance = vec3(1, 0, 0);
    EmitVertex();

    gl_Position = gl_in[1].gl_Position;
    color_gs = color_tes[1];
    triDistance = vec3(0, 1, 0);
    EmitVertex();

    gl_Position = gl_in[2].gl_Position;
    color_gs = color_tes[2];
    triDistance = vec3(0, 0, 1);
    EmitVertex();
    EndPrimitive();
}
```

In the above geometry shader, the `triDistance` attribute will be interpolated across each fragment generated by this primitive during rasterization. Thus, this `triDistance` corresponds to how close the generated fragment is to the edges of the primitive. This can be taken advantage of in the fragment shader to generate edges to the tesselation.

```
#version 400

flat in vec3 color_gs;
in vec3 triDistance;

out vec4 color;

void main(){

    float d1 = min(min(triDistance.x, triDistance.y), triDistance.z);
    if (d1 < 0.02) {
        color = vec4(0, 0, 0, 1);
    } else {
        color = vec4(color_gs, 1.0);
    }
}
```

16.4 The Final Step

Finally, we need a couple of things in our main C/C++ program to make our shaders work correctly.

- `glPatchParameteri(GL_PATCH_VERTICES, N);` must be called, where `N` is the number of vertices in the TCS.
- you must use `GL_PATCHES` as the primitive type in your calls to `glDrawArrays` or `glDrawElements`