Interactive Computer Graphics
Coursework – Task 4 (*assessed)

Bernhard Kainz, Antoine Toisoul, and Daniel Rueckert

February 9, 2017

Task 4: Geometry

Task 4a: Mesh subdivision*

In this exercise you will learn how to generate primitives within a geometry shader. So far, you have learned how to use a vertex shader and a fragment shader. In this task you will modify the *per-polygon* geometry shader. **To activate this shader in your rendering pipeline you need to tick the box beneath the geometry shader tab and rebuild the shader program** (`F5`)!

While it is possible to manipulate the position of incoming vertices in the vertex shader, a geometry shader is additionally able to emit new primitives (*i.e.*, vertices) into the pipeline and to transform them into different types.

In the following you will implement a very simple mesh subdivision algorithm as it is outlined in Figure 1.

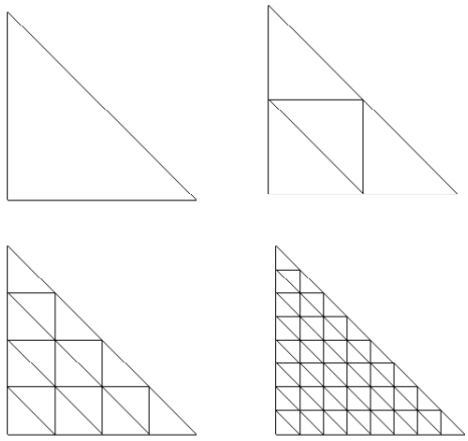
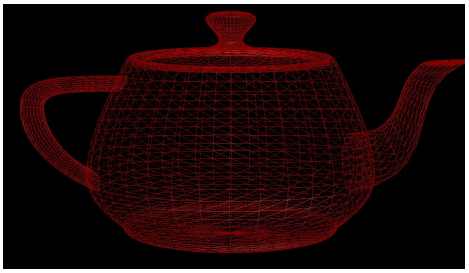


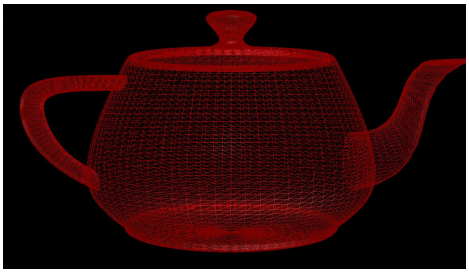
Figure 1: Subdivision on the example of a single triangle.

For this task you should use *Wireframe* mode to see the result of your computation. (*Scene* → *Enable wireframe*) To define the desired number of levels for the primitive subdivision you should use a `uniform` variable. An example of the output (without subdivision) is shown in Figure 2a.

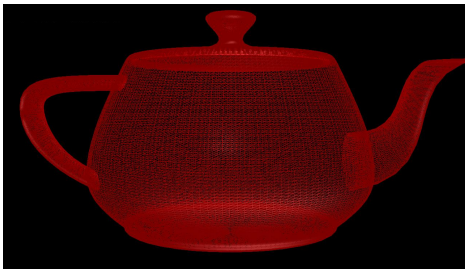
You could implement the subdivision using barycentric coordinates or simple vector arithmetic within a nested for-loop and use the functions `EmitVertex()` to generate a new vertex and `EndPrimitive()` to close the



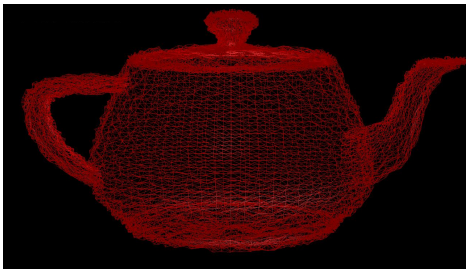
(a) Level 0



(b) Level 1



(c) Level 2



(d) Level 2 Bonus

Figure 2: Results of Exercise 3

new triangle. You can also define new functions, e.g. for producing a new vertex in barycentric coordinates similar to a simple C program. Do not forget the also interpolate the new vertex's normal vector. A pass-trough shader is already implemented in the *per-polygon* geometry shader, which shows the basic use of these functions:

Results for the subdivision are shown for one level in Figure 2b and for two levels in Figure 2c Note that the number of possible additional primitives that can be emitted by a geometry shader is limited and hardware-dependent. Therefore, you should limit your number of levels to 2 or 3.

Task 4b: Vertex animation

Optionally you may implement vertex animation at the geometry shader stage. Therefore you can choose any time-dependent displacement of the resulting vertex in direction of it's normal vector. To help your with this task, we provide a uniform variable `time` in the geometry shader, which is set by the host program to the current time since start of the shader program. You can also use pseudo-number generation functions initialized by, e.g. the vertex xy position, similar to:

```
float rnd(vec2 x)
{
int n = int(x.x * 40.0 + x.y * 6400.0);
n = (n << 13) ^ n;
return 1.0 - float( (n * (n * n * 15731 + 789221)
+ 1376312589) & 0x7fffffff) / 1073741824.0;
}
```

A snapshot of the animation may look like shown in Figure 2d. You can implement any animation you like, for example, a melting teapot.

HAVE A LOT OF FUN!!