CSc 471 - Fall 2019
Fundamentals of Computer Rendering
Assignment 4
University of Victoria

Due: November 10, 2019 at 23:55. **Late assignments will not be accepted.**

1 Overview

In 2007, NVidia published a paper describing a technique for rendering a wireframe on top of a shaded model using geometry shaders. The use of geometry shaders is significant since geometry shaders are the only stage in the pipeline that has access to the entire geometry (and can therefore either change or add information for future stages of the pipeline), The technique described in the NVidia paper used the geometry shader to compute the minimum distance from each vertex of the triangles to the opposing edge. This distance was then used by the fragment shader to determine the distance from the fragment to the edges of the triangle. If the fragment falls below the width of the wireframe, the colour of the fragment is mixed with the colour of the line. Otherwise, the fragment is rendered using the normal colour.

Another common rendering technique uses the tessellation shaders to increase the quality of a mesh which is initially specified using a small number of vertices. These are then forwarded by the vertex shader into the two tessellation shaders, which then subdivide the triangles to increase the quality of the mesh without impacting performance.

Your task for this assignment is to write a single C++ program that does the following:

- Renders an icosahedron to the screen,
- Uses tessellation shaders to subdivide the icosahedron and approximate a sphere (called a *geodesic*),
- Uses geometry shaders to render the wireframe on top of the shaded model.

Further implementation details are given in Section 4. In order to successfully implement the assignment, you should read the OpenGL Programming Guide, 9th Edition. The book is freely available through the UVic Library (and a direct link through the library's portal has been posted on conneX). Specifically, you should read and understand the following two chapters:

1. Chapter 9: Tessellation Shaders. In particular, how to render a mesh using tessellation shaders. Also, make sure you understand the built-in variables that tessellation control and evaluation shaders have, along with their layouts.

2. Chapter 10: Geometry Shaders. Make sure you understand the input and output of geometry shaders, how to emit vertices, and their layout. The shaders required for this assignment are very simple, but it is important to understand how to write them.

2 Specification

For full marks, your submissions **must** have the behaviour specified below. Failure to implement this functionality will result in marks being deducted.

- Accurately renders the icosahedron to the screen.
- Uses tessellation shaders to subdivide the icosahedron into a geodesic.
- Uses geometry shaders to render the wireframe on top of the shaded model.
- Implements controls to change the tessellation levels.
- Maintains a framerate of *at least* 60FPS as specified by the in-program counter.

You will be provided with an assignment bundle that will contain the following:

1. A full CMake setup for the assignment including code to generate the headers like the one seen in the lab,
2. an OBJ file for a torus knot,
3. a `main.cpp` file with a function to create the icosahedron vertices, as well as a simple UI with sliders for the tessellation levels and an FPS counter,
4. a `assignment.hpp` header containing some function definitions, and
5. a blank `UniformLocations.glsl` file.

Your submission must contain the following:

1. One `main.cpp` containing all the appropriate C++ code to perform the specified tasks.
2. One `assignment.hpp` containing the definitions of any auxiliary data structures, functions, etc that you may require. Note that you may leave this file unmodified.
3. Any vertex and fragment shaders that you require, along with the `UniformLocations.glsl` file (you may leave this blank if you wish). Note that these *must* observe the file extension convention specified in the labs.

Please note that the full assignment bundle does not need to be provided in your submission, only the individual files listed above.

3 Evaluation

Your code must compile and run correctly in ECS 354. If your code does not compile or crashes at runtime, you will receive a mark of zero. This assignment is worth 8% of your final grade and will be marked out of 10.

4 Implementation Details

4.1 First Step

The first stage of this assignment is to render the icosahedron to the screen. Start by creating a simple vertex and fragment shader as we have done in the labs. The vertex shader will convert the vertices of the icosahedron using the MVP matrix, while the fragment shader will just compute the final colour of the fragments. For this assignment, we will be using the Phong shading model that was used in assignment 1 without the specular component. The shading equation then becomes:

$$c_f = c_a + |N \cdot L| c_d$$

Where c_f is the colour of the fragment, c_a is the ambient colour, N is the normal of the surface, L is the light position, and c_d is the diffuse colour. Note that the icosahedron data does not contain normals. As such, this first stage will only use the ambient colour. Once you have this implemented, you should see something similar to Figure 1. Once you have this, the next step is to start implementing the tessellation shaders.

For this example, the following values are used:

Name	Value
Light position	$(0.25, 0.25, 1.0)$
Diffuse colour	$(0.04, 0.04, 0.04)$
Ambient colour	$(0.0, 0.75, 0.75)$

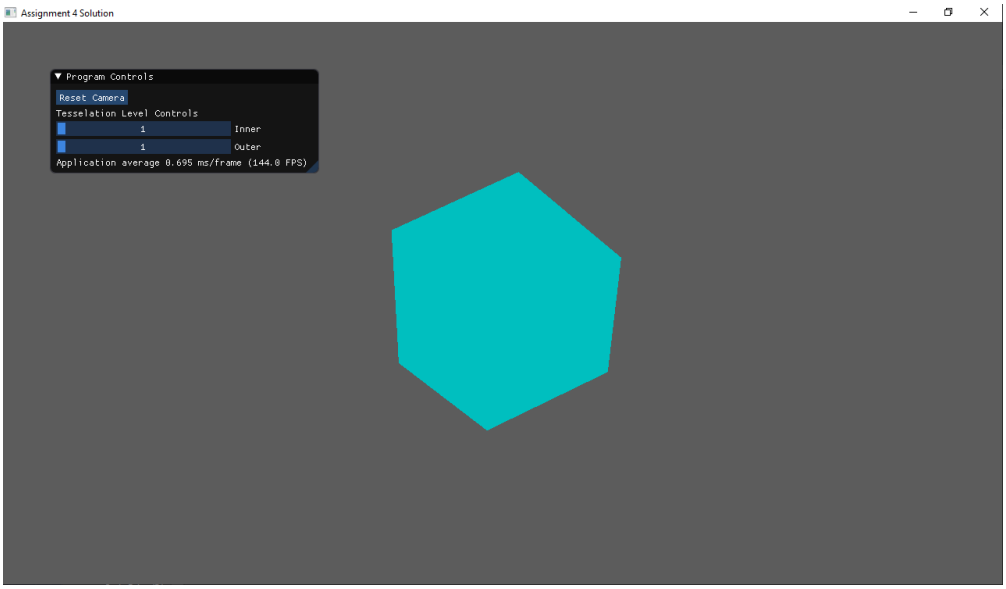

Figure 1: First stage.

4.2 Tessellation Shaders

So far we have used `GL_TRIANGLES` as our rendering primitive, but for tessellation shaders, we must change the primitive type from `GL_TRIANGLES` to `GL_PATCHES`. Once we have that, we can now get

started on the two components of the tessellation pipeline: the tessellation control shader and the tessellation evaluation shader.

4.2.1 Tessellation Control Shader

First modify your vertex shader to simply forward the provided vertex position. Note that the vertex shader will *not* write to `gl_Position`. It will simply write the vertex coordinate without any transformations into an output variable. Once we have this ready, we can begin the tessellation control shader. The role of this shader is twofold:

- First, it needs to forward yet again the position of the vertices to the tessellation evaluation shader.
- Second, it needs to specify the values of the inner and outer levels of tessellation through the built-in GLSL variables.

A few things to note here: first, note that we are dealing with triangles, hence the number of vertices in the layout of the tessellation control shader must be 3. Next, the input of the shader is no longer a single variable, but rather an array. This can be specified by `in vec3 inVar[]`. Similarly, the output of the shader will also be an array. The arrays will be indexed using `gl_InvocationID`. With this ready, we can now write the tessellation evaluation shader.

4.2.2 Tessellation Evaluation Shader

The tessellation evaluation shader will take input vertices from the control shader and set them to their correct position. Before we get into this, we need to specify the input layout of the shader. We are generating triangles that are equally spaced and have a clockwise winding. The input vertices are still in an array, and we will output two variables for our geometry shader: the position of the vertices and the patch distance. With this out of the way, we can now look at the tasks that the evaluation shader must accomplish:

- First, we need to take each component of the patch coordinates (provided in `gl_TessCoord`) and multiply them by the vertex positions to convert them from barycentric coordinates back to normal coordinates.
- Once we have the resulting vectors, we are now ready to position the vertices in their new coordinates. Since we are constructing an approximation of a sphere, then we can push every vertex along the normal of the sphere to place them. We compute the new position as follows: $|p_0 + p_1 + p_2|$ where p_0, p_1, p_2 are the vectors we computed in the previous step and $|\ |$ is the normalization operator.
- We are now ready to write to our output variables: the vertex position is the one we computed in the previous step, while the patch distance are the barycentric coordinates of the patch provided by `gl_TessCoord`.
- The last task is to finally write to `gl_Position`. This value is the vertex position we computed multiplied by the MVP matrix.

We are now done with the tessellation component. Make sure that everything compiles and links correctly before proceeding to the geometry shader.

4.3 Geometry Shaders

The role of the geometry shader is to compute the distances between each vertex and the opposing edge, as well as creating the triangles. Note that we are not modifying the triangles in any way, we are simply emitting them as they are. The input layout of the geometry shader is triangles. The output layout is a triangle strip, with a maximum number of vertices set to 3. The inputs are the position of the vertices as well as their distance. The outputs of the geometry shader are:

- The normal of the triangle,
- The distance from each vertex to the opposing edge,
- The triangle distance.

With this in mind, we can now begin implementing the shader. The first thing we will do is to compute the following expressions:

$$\begin{aligned} A &= p[2] - p[0] \\ B &= p[1] - p[0] \\ N &= |A \times B| \\ N &= \left(M^{-1}\right)^T \cdot N \end{aligned} \tag{1}$$

Where p is the input vertex position, and M is the model matrix. The final matrix operation is to correctly transform the normal of the triangle with the inverse transpose of the model matrix. With this done, N is now the normal of the triangle.

We are now ready to emit the vertices. For each vertex we emit, the distance from each vertex to the opposing edge are the barycentric coordinates that we received from the tessellation shaders. The triangle distance will be used in the fragment shader to highlight the triangles that belong to the original icosahedron. Because the triangles are equilateral, the distances are all unit and can be expressed with a vector of value $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ for the first, second, and third vertex respectively. The value that is written to `gl_Position` is taken from `gl_in[i].gl_Position` where i is the index of the vertex we are emitting.

The last step is to modify the fragment shader to use the new inputs from the geometry shader. Recall that the colour computation is:

$$c_f = c_a + |N \cdot L| c_d$$

Where N is the normal provided by the geometry shader. Once we have the colour, all that is left to do is to compute the colour of the edge and blend the two colours together. First, we will compute the distance of the fragment to the tessellated edges, and then to the original triangles from the icosahedron. This is computed as follows:

$$\begin{aligned} d_1 &= \min(\min(b_x, b_y), b_z) \\ d_2 &= \min(\min(t_x, t_y), t_z) \end{aligned} \tag{2}$$

Where b is the barycentric coordinates (distance from the vertices to the edges) and t are the triangle distances. Once we have this, we can compute the colour of the lines with a simple function

that amplifies the values of the lines. The function does the following:

$$\begin{aligned} d &= s \cdot d + o \\ d &= clamp(d, 0, 1) \\ d &= 1 - 2^{-2 \cdot d^2} \end{aligned} \tag{3}$$

Where d is the distance, s is a scale factor, and o is an offset. This function is called with both d_1 and d_2 as the distance. For this example, $s = 40$ for d_1 and $s = 60$ for d_2. In both cases, $0 = -0.5$. Let a_1, a_2 be the results of the amplify function when used with d_1, d_2 respectively. The final colour is then: $a_1 \cdot a_2 \cdot c_f$.

To test if your implementation works, set the inner and outer tessellation levels to 4. You should get an image similar to Figure 2.

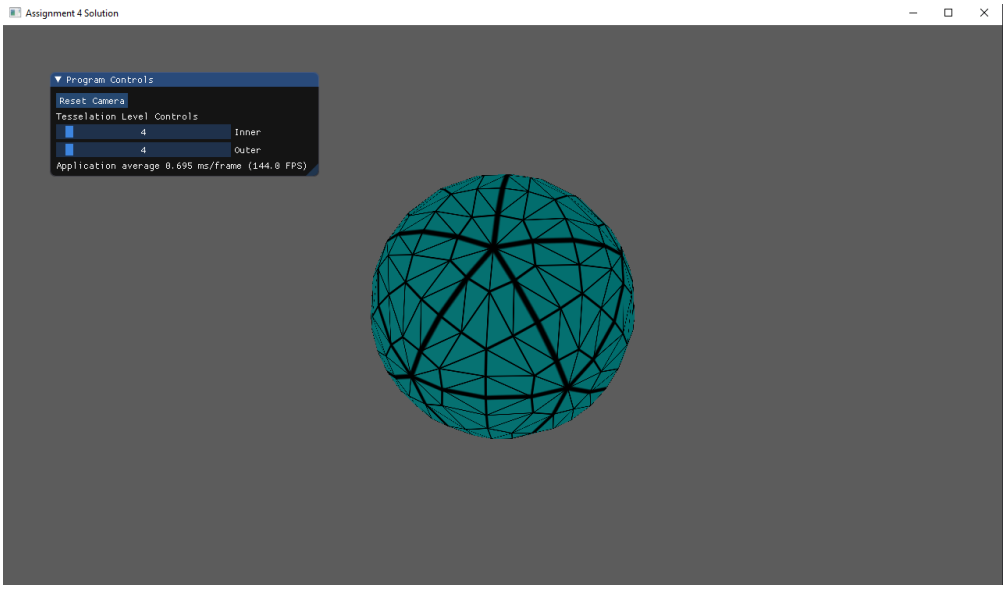

Figure 2: Geodesic with 4 as inner and outer tessellation levels.