Exam Project – Tessellation Shaders

Graphics Programming
KGGRPRG1KU

Oliver Astrup
olas@itu.dk

1 Introduction

This paper will give an overview of what tessellation is and what it can be used for in within the field of graphics programming. It also describes how to implement tessellation shaders using OpenGL. This paper will also discuss and show how this was implemented in the project.

The first section will give a broad overview of tessellation, by explaining several elements that are independent from implementation and what tessellation is used for. The second section will go more in-depth with the OpenGL library and how tessellation shaders can be implemented using this specific library. Then the third section is for the project, that was made together with this paper, it contains the implementation of a tessellation shader using OpenGL and the section explains how this was done and what it can do. The last two sections are reserved for a conclusion on the project and paper and what the next steps to improve it would be.

2 What is Tessellation?

Tessellation is the process of subdividing surfaces into smaller polygons, that can then be manipulated based on different variables. The point of using tessellation is to allow for the use of polygons with lower vertex counts to increase their complexity during rendering and thereby save resources. This technique is also used for implementing level of detail in scenes, so models further away from the camera will stay less detailed, without impacting the quality of the scene, as they would be too far away to be properly seen by the camera [1]. An example of this can be seen in figure 1.

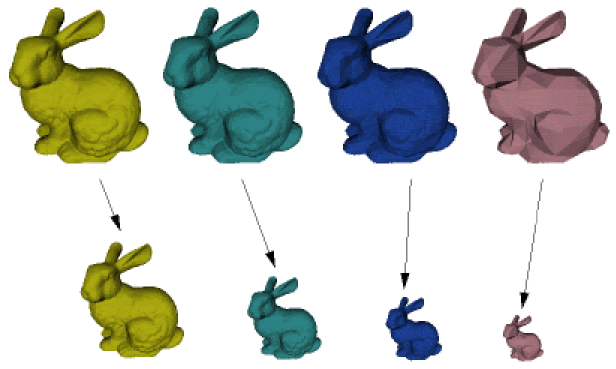
Figure 1: *Example of level of detail*

Another use of tessellation is during the use of expansive meshes that expands over a large area. In cases like this the mesh can be divided into different regions and regions closer to the camera will need to be more detailed than regions further away. Those will have lower, or no tessellation applied to them. An example of this is shown in figure 2, where the grass and the tree closer to the camera is more detailed than the same elements on the hill a bit further back in the scene. Then with the mountain in the background, the details are even lower, though the trees and grass can still be seen.

It is important to note that all the calculations done using tessellation is handed by the graphics hardware [4]. This means that there will be a reduced load of system resources due to lower number of calculations and storage need because of the lower number of stored vertices.

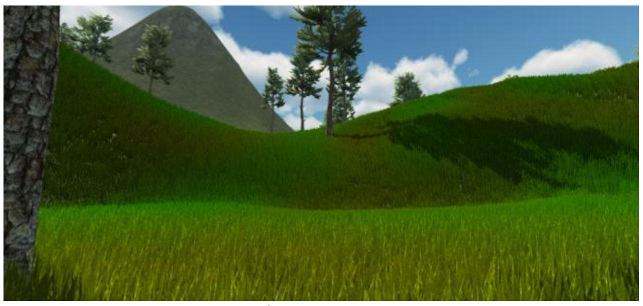
Figure 2: *Example of expansive mesh being divided.*

One last interesting technique that is available due to tessellation is to move the vertices along their normal to create offsets in a model's surface. This can be used to make flat surfaces more interesting by moving some of the vertices to make the surface appear to be rugged. This can be done in connection to level of detail, where the closer a camera is to the surface, the more rugged it becomes. This is displayed in figure 3.

3 Tessellation in OpenGL

In OpenGL tessellation is one of the steps in the rendering pipeline and is run right after the vertex shader. Though unlike the vertex shader, the tessellation step is completely optional and consists of three different stages: The Tessellation Control Shader, Tessellation Primitive Generator and Tessellation Evaluation Shader. To make use of tessellation in OpenGL only the Evaluation Shader is needed, as OpenGL can use a default Control Shader that just copies over information, though both are programmable, just like the vertex and fragment shaders [2].

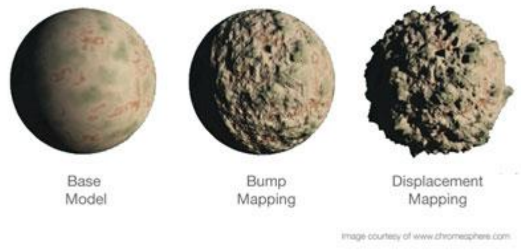

Figure 3: *Example of offsets in surfaces.* [4]

2.1 Patches

An important part of tessellation in OpenGL is the use of *patches*. These are primitives that consists of several control points (vertices), that will be used to determine the attributes of the new vertices. Figure 4 from the project shows a simple example of this, by having a single triangle with three vertices and a patch size of three. Resulting in all the vertices being used to determine the position and color of the new vertex. In this case the new point is an average of the it's control points, but that could change depending on implementation.

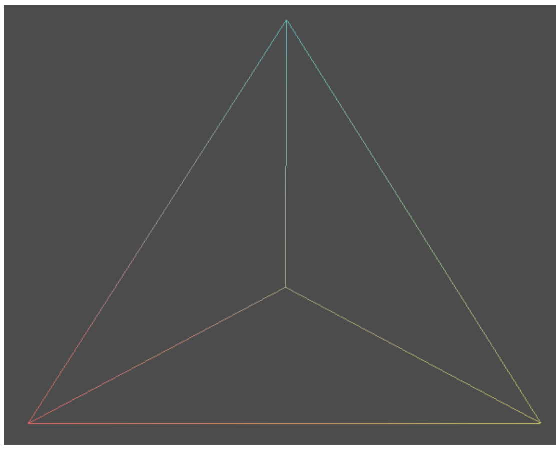

Figure 4: *Example of simple tessellation from project*

The important thing to notice about patches is that their size is depended on the implementation. The graphics hardware determines what the maximum possible amount of control points in a single patch can be, but it will always be at least 32 [2]. This allows for some huge primitives for better fine tuning of tessellation levels by using better interpolation methods like bicubic interpolation.

Another type of patch in OpenGL is the *abstract patch*. These patches are generated by the primitive generator, depending on the input expected by the evaluation shader. An abstract patch defines the connections of the vertices after tessellation has been applied. These can either be triangles as seen in figure 4, quads or isolines, depending on implementation.

2.2 Control Shader

The first part of tessellation in OpenGL starts in the control shader. It is an optional shader and is there to provide information about tessellation levels and any transformations to patches [2]. Since patches are not clearly defined as polygons, but instead as a surface described by these points, any changes made to a single control point, will change to the whole structure of the surface, though this is not always needed and can be omitted [3]. The other attribute of the Control Shader is to provide tessellation levels, these values determine how much tessellation should applied to the patch. There are two different tessellation levels, inner and outer, they both have a specific purpose. The outer levels are used to determine how the outer edges of the abstract patch should be divided, so if an edge has an outer tessellation level of 4, it will divide the edge into four smaller edges by creating three new vertices along it. This is displayed in figure 5, where the left most edge has been split into four distinct parts. It should also be noted that the bottom edge has an outer tessellation level of one, resulting in no tessellation to be applied.

The inner tessellation level is a bit more unintuitive; it is used to describe the number of times the space should be subdivided inside the figure. In figure 5, the triangle has an inner level of five, resulting in two new triangles. If the level was lowered to four, the inner most triangle would be just a single vertex instead and if the level was six, then a new vertex would appear in the middle of the inner most triangle.

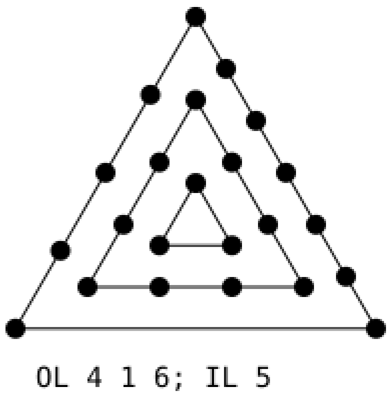

Figure 5: *Visualization of tessellation levels.* [2]

2.3 Primitive Generation

With the input from the control shader, the primitive generator preforms the actual tessellation of the models. There are several predefined variables that determine how this stage works, the tessellation levels, spacing, output type and winding order. The tessellation levels were described in section 2.2 and output type (abstract patch) were described in section 2.1.

For spacing there are three different types, equal spacing, fractional odd spacing and fractional even spacing. Equal spacing is the simplest form and is the spacing used in figure 5. It defines that all the edges generated by the subdivision should be of equal length. The other two types can be used to generate more smoother behavior during tessellation, as they will just divide edges by a set amount but also change the length of each of the segments. It works with two different values, an effective tessellation that is generated by rounding and used to calculate the number of subdivisions and a factional value, that is the value before rounding. Using these values, the spacing between the different points are calculated, the outer most edge will be of different length to the others, as it will be shorter depending on the difference between the effective tessellation and the fractal value. The difference between odd and even spacing is in the rounding methods used, both methods round up, but odd will round up to the nearest odd value and even to the nearest even value. [2]

For the winding order, this is simple stated as either clockwise or counterclockwise, so that it can be used in later stages for effects like culling of the patch. The output of the primitive generation is abstract patches that are send to the evaluation shader. It should be noted that each patch is defined by barycentric coordinates, this becomes important during the evaluation.

2.4 Evaluation Shader

The final part of tessellation in OpenGL takes part in the evaluation shader, this works much like the vertex shader, the difference being that it also transforms all the new vertices generated by the primitive generator and needs to place these in world space.

As described above the abstract patches from the primitive generator are described in barycentric coordinates, this displayed in figure 6. The newly generated vertices will have a position vector also in this coordinate system, so to generate an even tessellation, it position can be calculated by multiplying the position vector with each of the control points for the abstract patch. The same calculations can be used for the generation of colors, texture coordinates and/or normal vectors as these can also be part of the information stored in the control points.

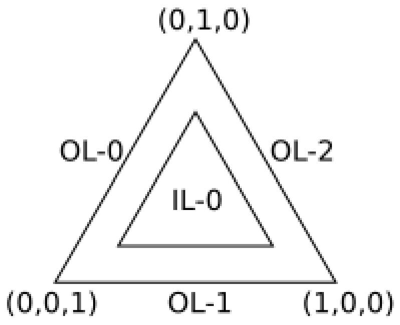

Figure 6: *A display of the architecture for a triangle.*

It is also in this shader that offsets can be applied by making use of different techniques like having a displacement map or just moving points along with their normal vector.

It should be noted that where the view projection that would normally be applied in the vertex shader, it should instead be applied in the evaluation shader, since all the newly generated vertices also should be multiplied by it and having it in both shaders would apply the view projection twice to the control points of the patches.

3 Project

This project has been focusing on implementing tessellation in OpenGL by programming both a tessellation control shader and a tessellation evaluation shader. It has been tested on a NVIDIA GeForce RTX 2070 graphics card and are using OpenGL 4.0 as this is the lowest version needed for implementing tessellation [2]. Much of the codebase is reused from hand ins and exercises throughout the course and therefore only code that is relevant for tessellation has been commented and code that have new or major changes from the original.

A focus point of the project has been to not hardcode the tessellation variables, but instead give the user the ability to manipulate them to allow for a better understanding of tessellation. Therefore, the program contains a menu, that can be opened using the spacebar, that allows the user to modify different values like tessellation levels and the use of level of detail, to see the effect tessellation can have on a model. The project and menu can be seen in figure 7.

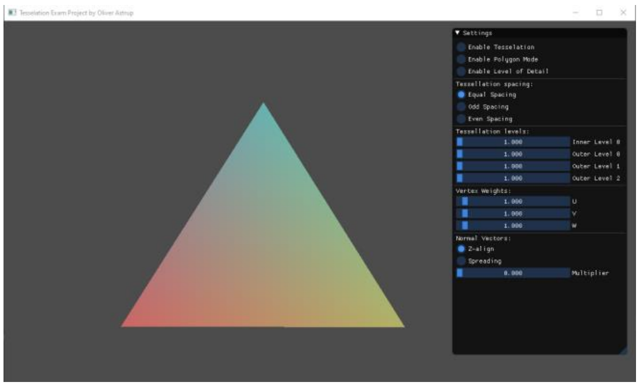
```
▼ Settings
○ Enable Tesselation
○ Enable Polygon Mode
○ Enable Level of Detail
Tessellation spacing:
● Equal Spacing
○ Odd Spacing
○ Even Spacing
Tessellation levels:
1.000   Inner Level 0
1.000   Outer Level 0
1.000   Outer Level 1
1.000   Outer Level 2
Vertex Weights:
1.000   U
1.000   V
1.000   W
Normal Vectors:
● Z-align
○ Spreading
0.000   Multiplier
```

Figure 7: *A display of the project with default settings.*

The program makes use of a simple triangle model to show the effects of the different tessellation techniques in real time. At the start default values has been selected as the least complex calculations and techniques. A combination of vertex shader and fragment shader, without tessellation shaders have also been added to compare with the changes that tessellation applies. A polygon mode has also been added to better show the effects of tessellation without having to transform the positions using normal vectors, as a flat surface will be visually alike with or without tessellation. This is what is displayed in figure 4.

3.1 Level of Detail

The program has implemented a simple level of detail technique that will change the tessellation levels in the control shader depending on the distance from the camera. This is just a simple implementation and does not make use of normal vectors, that could help with showing more depth in the model as described in section 1.

The inner tessellation level is determined by the highest outer level, as this would describe the players distance from the model.

3.2 Tessellation Spacing

As described in section 2.3 there are different tessellation spacing types. This implementation contains all the different types with the equal spacing selected by default. Because of spacing needed to be defined when evaluation shaders are implemented, there was a need to have three different evaluation shaders, one for each spacing type. Other then that, the shaders are identical. Figure 8 displays the effects of even spacing, where the triangles are of many different sizes.

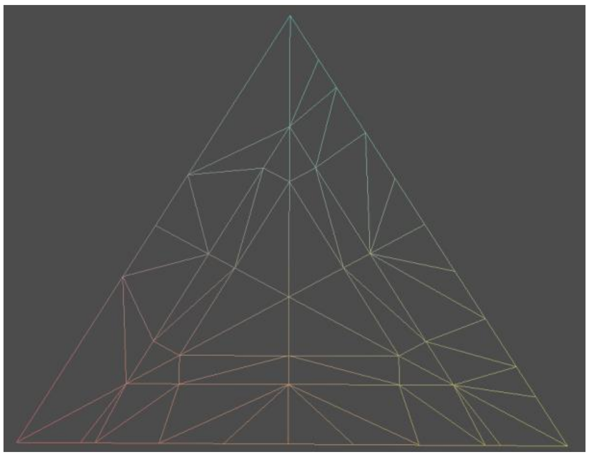

Figure 8: *A display of even spacing*.

3.3 Tessellation Levels

Section 2.2 has a description of the tessellation levels. The program allows for the user to manipulate all the inner and outer levels available to better study the effect of each of them. Note that this does not work with level of detail, as that will override the levels with the dynamically calculated levels. Because fractal spacing can use decimal values, all the levels are saved as floating-point numbers. Though OpenGL automatically rounds up levels when doing equal spacing, so that is not a problem.

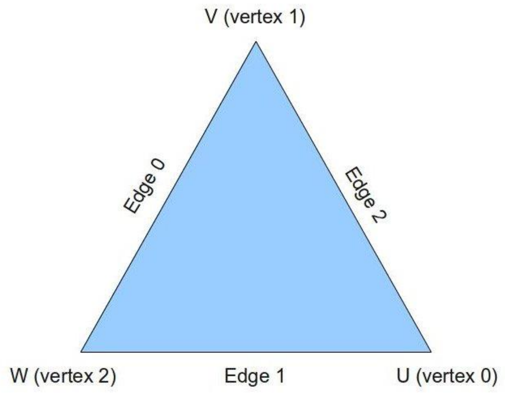

Figure 9: A display of edges and control point naming. [3]

3.4 Vertex Weights

These weights have little to do with tessellation but are added in the evaluation shader, to be able to make more disformed figures, to show that there is a lot of control to be had when positioning vertices and they do not have to be placed in the perfectly central of the patch. Each of the values correspond to a specific vertex in the barycentric coordinates and follows the structure of figure 9. Steps have been taken during the implementation to make the overall triangle displayed just as this figure, to make it easier to follow for a user.

3.5 Normal Vectors

The last part of the program is focus on the use of normal vectors to displace vertices on the surface. Two different sets of normal vectors have been implemented for the model, the first to check if normal vectors worked correctly and the second one to show of what normal vectors can be used for. A multiplier has also been added here to control how much a vertex should follow their normal vector. This is just a very simple calculation with the normal vector multiplied with the value being added to the vertex position.

The first set is called *Z-align* and have all the control points implemented with normal vectors along the Z-axis. This just makes the model move back and forth in space without showing many properties. Though it shows that all vertices generated by tessellation moves perfectly along with this as there is no displacement.

The second set is called *Spreading* and have the normal vectors of the control points in different directions. This will bring out displacement in the model, as the interpolations of tessellated vertices now have different normal vectors and will move around in space differently. The effect of this is shown in figure 10. It can also be seen from this figure that now the changes from tessellation can be seen without the use of the polygon mode, since it’s no longer just a flat surface.

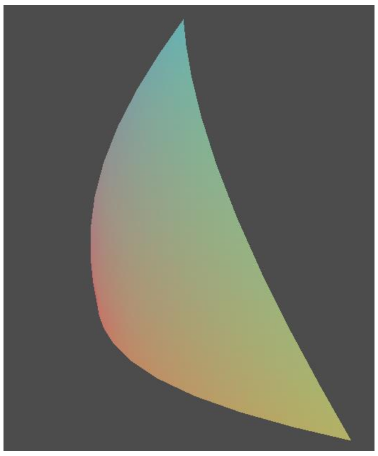

Figure 10: *A display of a curved model made following the interpolated normal vectors.*

4 Conclusion

Overall, the project has achieved what it was meant to do, by implementing both a tessellation control shader and a tessellation evaluation shader, both with multiple values that could be modified during the runtime of the application. Though some of the more advanced aspects of tessellation could have been explored more in the implementation and the implemented techniques could have been tested on more complex models.

The other aspect of this project was to acquire knowledge on a topic within the subject of graphics programming. This report contains all this knowledge by explaining both what tessellation is and how it works within the specifics of OpenGL. Of cause other uses of tessellations are available, though they will build upon the foundation of what has been described in this paper.

5 Future Work

As stated in the conclusion above there are still elements of the theory that have not been implemented into the project itself. Though it all builds on the elements that have been implemented already.

In section 2.1 the different abstract patch types have been described. Though both quads and isolines have not made it into the project. Since these aspects are being explicitly described in the evaluation shader and with the already heavy use of three different shaders to handle spacing. Each of these would have the same problem and would result in the use of nine different evaluation shader. Though reading I have also not been able to find a proper use of iso-lines. Except a single paper about the use of tessellation for creating splines [5].

While there are differences between the different abstract patch types, they work in much of the same way and should be quite easy to understand, with knowledge about how the triangle patch works.

The project also only works on a single triangle with three pre-defined vertices used for control points, this works well for illustrating the different aspects available in the project. Though it would have been fascinating to also explore other models, some with just a few more vertices and some could be quite expansive. This would also have opened for exploring the different patch sizes and what could be done to patches in the control shader. Though again I have been unable find papers describing how to do this in practice.

The last element to explore would be to work more with the normal vectors. Unlike the previous aspects of this section, this is a very central part of tessellation. Multiple of the sources linked to in this paper makes use of normal vectors to make their models look less flat, e.g., though displacement maps.

References

[1] Gregory, Jason. 2019. *Game Engine Architecture*. 3rd Edition CRC Press. Chapter 11.

[2] Tessellation (OpenGL Wiki). [ONLINE] Available at: https://www.khronos.org/opengl/wiki/Tessellation [Accessed 03 January 2022]

[3] Basic Tessellation. [ONLINE] Available at: https://ogldev.org/www/tutorial30/tutorial30.html [Accessed 04 January 4, 2022]

[4] Burke, Steve. 2015. *Defining Tessellation and Its Impact on Game Graphics (with Epic Games)* [ONLINE] Available at: https://www.gamersnexus.net/guides/1936-what-is-tessellation-game-graphics [Accessed 05 January 2022]

[5] Rasterization of Parametric Curves using Tessellation Shaders in GLSL. 2015. [ONLINE] Available at: https://computeranimations.wordpress.com/2015/03/16/rasterization-of-parametric-curves-using-tessellation-shaders-in-glsl/ [Accessed 05 January 5, 2022]