PN-triangle tessellation using Geometry shaders

The effect on rendering speed compared to the fixed function tessellator
This thesis is submitted to the Department of Interaction and System Design at Blekinge Institute of Technology in partial fulfillment of the requirements for the Bachelor degree in Computer Science. The thesis is equivalent to 10 weeks of full time studies.
Contact Information:

Author:
Martin Löwgren
Address:
Studentvägen 1:09, 372 40 RONNEBY
E-mail:
baldrik@gmail.com

Author:
Niklas Olin
Address:
Fogdevägen 4b, 371 40 KARLSKRONA
E-mail:
olin.niklas@gmail.com

University advisor:
Stefan Petersson
Department of Software Engineering and Computer Science
Address: Soft Center, RONNEBY
Phone: +46 455 385731

Department of
Interaction and System Design
Blekinge Institute of Technology
SE - 372 25 RONNEBY
Sweden
Internet: http://www.bth.se/tek/ais
Phone: +46 457 38 58 00
Fax: +46 457 271 25
Abstract

With each computer game generation there is always a demand for more visually pleasing environments. This pushes game developers to create more powerful rendering techniques and game artists to create more detailed art. With a visually stunning backdrop also comes the need for high-resolution models.

A common issue is that if all models in a scene are high-resolution it would not only require immensely powerful hardware, it would also be wasteful as only the models in the foreground are close enough that we would recognize the increased details. The common solution to this problem has been to load several versions of each model containing varying amounts of detail. However this solution has the drawback that it increases our memory footprints as more models are loaded into the memory. Tessellation offers a more dynamic solution to the problem as it only requires us to load a low-resolution model and higher resolution versions can be generated during run-time on the GPU.

With the introduction of DirectX 11 tessellation is now supported in the hardware, however we are still a few years away from seeing DirectX 11 being used as the core of any 3D rendering engine. In a transitional period like this between hardware generations game developers has to tackle the dilemma that the current hardware generation has to be supported when creating games that will also utilize the next generation. This thesis focuses on comparing the performance of a tessellation scheme supported by the current hardware generation, DirectX 10, as opposed to a scheme developed for the next generation, DirectX 11.

Two prototypes, one using the Geometry shader that was introduced in DirectX 10 and the other using the fixed function tessellator introduced in DirectX 11, were built to compare the performance of tessellated model rendering. Several different variants of each prototype were tested and the general conclusion is that the tessellator performed better than the Geometry shader.

Keywords
Tessellation, DirectX 10, DirectX 11, Geometry shader, PN-triangles, fixed function tessellator
Table of Contents

ABSTRACT ................................................................................................................................. 1-4
KEYWORDS ................................................................................................................................. 1-4

1 INTRODUCTION ......................................................................................................................... 1-1
1.1 BACKGROUND ......................................................................................................................... 1-1
1.2 RESEARCH QUESTION .......................................................................................................... 1-1
1.3 RESEARCH OBJECTIVES ..................................................................................................... 1-2
1.4 HYPOTHESIS ........................................................................................................................ 1-2
1.5 METHODOLOGY ..................................................................................................................... 1-2
1.6 DELIMITATIONS ...................................................................................................................... 1-2
1.7 ACKNOWLEDGMENTS ............................................................................................................. 1-2

2 DIRECTX .................................................................................................................................... 2-3
2.1 DIRECTX 10 PIPELINE ......................................................................................................... 2-3
2.2 INPUT ASSEMBLER STAGE .................................................................................................... 2-3
2.3 VERTEX SHADER .................................................................................................................... 2-4
2.4 GEOMETRY SHADER .............................................................................................................. 2-4
2.5 STREAM-OUTPUT STAGE ..................................................................................................... 2-4
2.6 RASTERIZER STAGE .............................................................................................................. 2-4
2.7 PIXEL SHADER ....................................................................................................................... 2-4
2.8 OUTPUT MERGER STAGE ....................................................................................................... 2-4
2.9 DIRECTX 11 PIPELINE ADDITIONS ....................................................................................... 2-5
2.10 HULL SHADER ...................................................................................................................... 2-5
2.11 FIXED FUNCTION TESSELLATOR ......................................................................................... 2-6
2.12 DOMAIN SHADER ............................................................................................................... 2-6

3 PN-TRIANGLES .......................................................................................................................... 3-7
3.1 PARAMETRIC CURVED SURFACES ....................................................................................... 3-7
3.2 PN-TRIANGLES ...................................................................................................................... 3-8
3.2.1 Barycentric coordinates ..................................................................................................... 3-8
3.2.2 Vertex control points .......................................................................................................... 3-8
3.2.3 Normal control points ........................................................................................................ 3-10
3.2.4 Performance ...................................................................................................................... 3-11
3.2.5 Visual result ....................................................................................................................... 3-11

4 PROTOTYPES ............................................................................................................................ 4-12
4.1 APPLICATION ......................................................................................................................... 4-12
4.2 TESSELLATOR PROTOTYPE .................................................................................................. 4-13
4.2.1 GPU Implementation ......................................................................................................... 4-13
4.3 GEOMETRY SHADER PROTOTYPE 1 ..................................................................................... 4-15
4.3.1 GPU Implementation ......................................................................................................... 4-15
4.4 GEOMETRY SHADER PROTOTYPE 2 ..................................................................................... 4-18
4.4.1 GPU Implementation ......................................................................................................... 4-18
4.5 BARYCENTRIC COORDINATE GENERATION ......................................................................... 4-18

5 PERFORMANCE TESTING ........................................................................................................ 5-20
5.1 TEST CASES .......................................................................................................................... 5-21
5.1.1 Case 1 – Single pass .......................................................................................................... 5-21
5.1.2 Case 2 – Focused emitting ................................. 5-21
5.1.3 Case 3 – Workload balancing ........................................ 5-21
5.1.4 Case 4 – Stored tessellation ........................................ 5-21
5.1.5 Case 5 – Fixed function tessellator ............................... 5-21
5.1.6 Case 6 – Fixed function tessellator + Geometry shader .... 5-21
5.2 TEST SCENARIOS .................................................. 5-22
5.2.1 Scenario 1 – One cube .............................................. 5-22
5.2.2 Scenario 2 – 20 Cubes ............................................... 5-22
5.2.3 Scenario 3 – One Wizard ........................................... 5-23
5.2.4 Scenario 4 – 20 Wizards ............................................ 5-23
5.3 TEST CASE CONCLUSIONS ...................................... 5-24
5.3.1 Test Case 1 Conclusions: ........................................... 5-24
5.3.2 Test Case 2 Conclusions: ........................................... 5-24
5.3.3 Test Case 3 Conclusions: ........................................... 5-24
5.3.4 Test Case 4 Conclusions: ........................................... 5-25
5.3.5 Test Case 5 Conclusions: ........................................... 5-25
5.3.6 Test Case 6 Conclusions: ........................................... 5-25
6 CONCLUSIONS AND FURTHER WORK .................................. 6-26
6.1.1 Discussion ............................................................ 6-26
6.1.2 Conclusions .......................................................... 6-27
6.1.3 Further work .......................................................... 6-28
7 REFERENCES ................................................................ 7-29
7.1 BIBLIOGRAPHY .......................................................... 7-29
7.2 WEBSITES .................................................................. 7-29
7.3 IMAGES ..................................................................... 7-29
8 APPENDIX A – TESSELLATOR PROTOTYPE ............................... 8-30
9 APPENDIX B – GEOMETRY SHADER PROTOTYPE ...................... 9-32
9.1 APPENDIX B1 – CASE 1.2 .............................................. 9-32
9.2 APPENDIX B2 – CASE 2 ............................................... 9-35
10 APPENDIX C – HELPER FUNCTIONS ................................... 10-40
1 Introduction

1.1 Background

As the demand for high quality 3D-scenes rises so does the need for highly detailed models. Game developers are able to go a long way towards reaching this goal using advanced lighting techniques such as parallax mapping and sub-surface scattering combined with rendering techniques such as normal mapping [MHH08]. However the need for increased geometric detail is still a factor as low-resolution models will always look coarse when viewed up close, mainly along the silhouette. Tessellation offers a solution to this issue.

Tessellation can be used in games to dynamically increase the resolution of a model without the need to generate several versions ahead of time. Increasing the number of triangles in a mesh using a tessellation scheme creates a result that smooth out the jagged edges normally seen in low-resolution models. This can be combined with other rendering techniques to further increase the detail of the model.

As DirectX 11 will not be a standard in graphic technology for a few years to come there is need for alternative solutions to using its fixed function tessellator when rendering a tessellated model. Another potential drawback to using the tessellator is that as it provides a general solution to handling tessellation for a vast range of subdivision levels it raises the question that this might limit its performance in specific cases. Our thesis aims at comparing the performance of a Geometry shader approach to tessellation with tessellation implemented using the DirectX 11 tessellator, in order to discern how well the tessellator actually performs in different scenarios.

1.2 Research question

The focus of this thesis is whether a Geometry shader approach would be a viable candidate to use when implementing a tessellation scheme.
1.3 **Research objectives**
This thesis aims to discern if tessellation using the fixed function tessellator will always outperform tessellation using a Geometry shader. Performance in this case refers to the average time to render one frame.

1.4 **Hypothesis**
*Tessellation using the Geometry shader will perform better than the tessellator for low subdivision levels.*

This hypothesis was reached from the fact that a general solution to a problem is on average slower than a solution tailored to handling a specific case.

1.5 **Methodology**
The approach chosen to benchmark the performance was to build two prototypes which were used to test different scenarios. In total we did six tests that each made some modifications to the core prototypes.

In order to get comparable results both prototypes were based on the same core application and all tests used the same tessellation scheme.

1.6 **Delimitations**
There are several different algorithms that can be used to tessellate a mesh, since this thesis focuses on the performance of different shader approaches and not the performance between algorithms it is limited to an implementation of the PN-triangle algorithm [Vlachos01]. This particular algorithm was chosen since it focuses on simplicity and efficient performance while still producing a visually aesthetic result.

Additional rendering techniques such as adaptive tessellation or displacement mapping will not be considered in this thesis as they would potentially compromise the results of the tests.

To fully understand this thesis, basic knowledge about 3D-programming using DirectX and linear algebra is required.

1.7 **Acknowledgments**
We would like to thank Stefan Petersson for helping us refine the idea for this thesis and providing feedback along the way. We also want to say thanks to Jonas Petersson and Veronica Axelsson for providing the 3D-models used in the test cases.
2 DirectX

This chapter will cover the DirectX 10 pipeline and the additions that were made to the pipeline with DirectX 11 that are relevant for this paper. This chapter will only describe each topic briefly to give the reader a better understanding of the pipelines, a more in-depth explanation can be found in [Luna08] and [MSDN].

2.1 DirectX 10 Pipeline

The DirectX 10 Pipeline offers the programmer three programmable shaders; Vertex Shader, Geometry Shader and the Pixel Shader, which are all compiled using shader model 4. There are also a couple of non-programmable stages; Input Assembler-, Rasterizer-, Output Merger- and Stream Output Stage. Although these stages are not programmable, it is still possible to configure them by setting different render states through the DirectX API. The relationships between the shaders and stages can be viewed in Figure 2.1.

2.2 Input Assembler Stage

As a draw call is made, the first stop for a mesh is the Input Assembler Stage. The objective of this stage is to assemble geometric primitives from vertex and index data that is stored in the video memory, depending on the primitive topology set from the DirectX API.

In DirectX 10, there are five basic types of primitives; Point list, line list, line strip, triangle list and triangle strip. All basic primitive types, except point list, can also be stored with adjacency data providing information about neighboring primitives. The difference between a strip and a list is the way the vertices will be bound together. When the primitives have been assembled, the vertices are fed into the Vertex shader.
2.3 Vertex Shader
The Vertex shader is the first programmable shader in the pipeline and is invoked once for every vertex point in the mesh. A triangle consists of three vertex points, and each of these points describes where two polygon edges meet. Apart from positional data a vertex point usually contains other information pertaining to that point, such as normal data and texture coordinates.

From here it is possible to add various effects, such as transformation or lighting, to each vertex point and then send it further down the pipeline.

2.4 Geometry Shader
The Geometry shader is optional and is invoked once for every primitive inputted to the pipeline, which can be either a point, line or a triangle, and each vertex point on the primitive can then be accessed in the shader.

The unique ability of the Geometry shader is that it gives the programmer the possibility to create or destroy geometry in a mesh.

Output from this shader differs from the Vertex shader as it outputs primitives through a stream, which does not have to be the same as the input primitive. This stream can be either a Point stream, Line stream or a Triangle stream and will output the primitives as a strip. In order to format the output as a list the RestartStrip function has to called after each complete primitive, e.g. three vertex points if the output is a triangle list, has been appended to the stream.

2.5 Stream-Output Stage
The Stream-Output stage allows the programmer to stream vertices to a buffer from a Geometry shader, giving the programmer the possibility to perform calculations on the GPU without drawing to the back buffer. Streaming can also be done from a Vertex shader if the Geometry shader is disabled.

2.6 Rasterizer Stage
The Rasterizer Stage is responsible for transforming the vertices from homogenous clip space to 2D-coordinates residing in the view port. The x and y-coordinates of the vertex point describe the position in the view port in units of pixels, while the z-coordinate normally remains untouched as it is used for depth testing. When this transformation is done all the vertex attributes has to be interpolated linearly for each pixel to get their correct values.

It is possible to configure the Rasterizer Stage by setting a Rasterizer state. Here one can decide if triangles should be back- or front-face culled, the winding order of the triangles and if the triangles should be drawn in solid or wireframe mode.

2.7 Pixel Shader
The Pixel shader operates on per pixel level and is responsible for calculating the color of a pixel.

2.8 Output Merger Stage
The Output Merger stage is the last stage before a pixel is drawn to the back buffer. Here two tests can be done to determine whether a pixel should be drawn or not, a depth test and a stencil test which can be configured by setting a Depth Stencil State through the DirectX API. The tests are done
using two buffers, a depth buffer and a stencil buffer, which has to be at least the same size as the back buffer.

The depth buffer stores the depth information of each pixel drawn to back buffer. The pixels in the depth buffer contains a floating point value ranging from 0.0 to 1.0, which allows the depth test to compare the current pixel's depth with the corresponding pixel's depth on the depth buffer to determine if the pixel should be drawn or not.

The stencil buffer stores a value for each pixel drawn to the back buffer, making it possible to flag pixels. This allows the stencil test to check against this buffer and determine if the pixel should be drawn to the back buffer or not. [Luna08] describes how the stencil buffer can be used to create different effects.

For both tests there are several settings that will affect how the tests are performed.

### 2.9 DirectX 11 Pipeline Additions

As of the introduction of DirectX 11, two new programmable shaders were added; the Hull Shader and the Domain shader, which are compiled using shader model 5. Along with these two shaders, a fixed function tessellator was also added.

With the new shaders working with different types of patches; quads, triangles or isolines, each patch having 1 to 32 control points, 32 new primitive topologies have been added called control point patchlist, one for each number of control points.

### 2.10 Hull Shader

The Hull Shader consists of two functions. One main Hull shader which performs calculations on each control point separately and one patch constant function that performs calculations on the entire patch. Before the Hull shader can be called, a few attributes needs to be set:

- [domain] – Specifies which type of patch the Hull shader will be working with.
- [partitioning] – Indicates how the tessellator will interpret the tessellation factors.
- [outputtopology] – For the tessellator to be able to create the right barycentric coordinates, it needs to be aware of which kind of primitives we want to deal with in the domain shader later on, which this attribute will tell us.
- [outputcontrolpoints] – Describes the number of control points that will be output from the Hull shader.
[patchconstantfunc] – Specifies the name of the patch constant function.

2.11 Fixed function tessellator

The fixed function tessellator generates barycentric coordinates depending on a specified subdivision level, which needs to be set and passed on by the patch constant function. The tessellator splits the subdivision description into two parts, inside tessellation factor and outside tessellation factor. With a uniform tessellation factor the tessellator produces a number of vertex points equal to the factor along the border and a number of vertex points consistent with subdivision level (factor – 2) to 1 residing inside the original triangle. How the tessellator iterates over the subdivision levels depends on the partitioning attribute, e.g. setting the attribute to fractional odd the tessellator will only create new points for odd subdivision levels.

2.12 Domain Shader

The Domain shader can be seen as a post-tessellation Vertex shader and is invoked for every barycentric coordinate that is generated by the Tessellator. It is responsible for creating new vertex points using the barycentric coordinates, the primitive control points from the main Hull shader, and the patch data received from the patch constant function. How to calculate the new vertex points using this data is described in the next chapter.

When tessellation is active the Geometry shader, which is located after the Domain shader, cannot use adjacency data in the mesh as it is invalidated by the recalculation of all vertex points.
3 PN-triangles

This chapter will cover the basics of the curved point-normal triangles scheme or PN-triangles for short, used in this thesis to create tessellated meshes. There exists several other ways to tessellate a mesh [MHH08], however these will not be covered as it is beyond the scope of this thesis.

3.1 Parametric curved surfaces

In computer game graphics we most commonly start out with a polygonal mesh consisting of vertex points that can be connected to form a number of flat triangles. A common issue when rendering a mesh is that if it consists of too few triangles any creases on the model can appear sharp, on the other hand if we increase the number of triangles rendering time is also affected adversely.

Tessellation is a scheme designed to increase mesh detail by replacing the flat triangles with a curved surface that are then re-triangulated into several smaller flat triangles during run-time.

In two-dimensional space smoothing can be done via a parametric curve. These curves are made up of a number of points and using repeated linear interpolation a smooth curve, as illustrated in figure 3.1, can be created. For a more in-depth examination of parametric curves see [MHH08].

In three-dimensional space the equivalent of a parametric curve is a parametric curved surface commonly in the form of a rectangular patch, known as Bézier patches. This surface is made up of a number of control points which can be evaluated to find any point along the curved surface. These points are connected to form a number of sub-triangles approximating the true surface. The reason that triangles are used to approximate the curved surface is that they can be efficiently rendered using the graphics hardware.

When rendered the surface of the sub-triangles are linearly interpolated, therefore more triangles gives a better approximation of the curved surface. The exact number of triangles that will be created from the curved surface is defined by the subdivision level, as described in 2.11.
3.2 PN-triangles

PN-triangles is a scheme created by Vlachos to create a more visually pleasing representation of any low-resolution mesh by substituting the triangles with PN-triangles. The layout of a PN-triangle is defined as a Bézier triangle, which is a parametric curved surface containing control points spread out over triangular grid. These control points are divided into two sets, one for positional data and the other for normal data. The control point calculation used in this chapter were developed by Vlachos, Peters, Boyd and Mitchell and described in [Vlachos01].

The focus of PN-triangle algorithm is simplicity; this is achieved by creating a curved surface that does not require neighborhood information or additional vertex data.

3.2.1 Barycentric coordinates

To obtain a point on the curved surface barycentric coordinates are used together with the control points. These coordinates can be seen as a numeric triple typically named \( w = (u, v, w) \), describing a weighted distance from the three corner points on the triangle. Recall that a point within a triangle \( \Delta p_0 p_1 p_2 \) can be described as \( p(u, v) = p_0 + u(p_1 - p_0) + v(p_2 - p_0) = (1 - u - v)p_0 + up_1 + vp_2 \), where \( u, v \) are barycentric coordinates and \( w = 1 - u - v \). The coordinates must fulfill a few requirements, namely \( u \geq 0, v \geq 0 \) and \( u + v \leq 1 \). Furthermore all points along the border of the triangle have one value that equals zero and all points inside the triangle all values must be a non-zero number.

3.2.2 Vertex control points

The goal of this scheme is, as previously stated, to create a number of new vertex points to improve the visual appeal of the model. The first step to creating these vertex points is to calculate the control points based on the existing triangles in our control mesh. The control points of a PN-triangle are defined as a cubic patch which in two-dimensional space equals a third-degree curve. This results in ten control points divided into three groups. \( b_{300}, b_{003}, \) and \( b_{030} \) are the corner control points and coincide with the original triangle. \( b_{201}, b_{102}, b_{012}, b_{021}, b_{120}, \) and \( b_{210} \) are tangent control points spread out along the border of the triangle, and \( b_{111} \) is the center control point that gives the surface its curved shape.
To calculate the border control points, the first step is to spreading them out along the border of the original triangle to get an intermediate position.

\[ b_{ijk} = (iP_1 + jP_2 + kP_3)/3 \]

One of the values \( i, j, k \) always equals zero ensuring that the coordinate is situated on the border of the triangle.

The goal is to create a curved surface while still maintaining continuity between the adjacent triangles. To solve this Vlachos, [Vlachos01], opted to use the vertex corners and their normals as these are values that adjacent triangles typically share. Using these values the intermediate position can be projected into the tangent plane defined by the corner normal. Recall that the projection of a point \( Q \) onto a plane with the normal \( N \) attached to a point \( P \) is \( Q' = Q - wN, w = (Q - P) \cdot N \). In this equation \( \cdot \) is used to denote the dot product. Merging both steps the complete formula for calculating the position of the border coordinates is as follows.

\[ b_{ijk} = \frac{(2P_x + P_y - w_{xy}N_x)}{3} \]

Here \( x \) represents the largest value of \( ijk \) and \( y \) represents the second largest.

Finally \( b_{111} \) is positioned at an intermediate position and readjusted using the average of all six border control points. The reason behind the following equation is to reproduce that of a quadratic polynomial giving us the curved shape, as shown in [Vlachos01].

\[ b_{111} = \frac{1}{4} (b_{210} + b_{120} + b_{102} + b_{201} + b_{021} + b_{012}) - \frac{1}{6} (b_{300} + b_{030} + b_{003}) \]

As mentioned earlier each point on a parametric surface can be found using repeated interpolation, it turns out that they can also be described as an algebraic function. For PN-triangles Bernstein polynomials are used to describe the points. The benefit of using this form is that all points will be located in the convex hull of the surface, meaning that they will stay “close” to the curve. The formula used to describe a PN-triangle using Bernstein form is as follows:

\[ b(u, v) = \sum_{i+j+k=3} b_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k \]

\[ = b_{300}w^3 + b_{030}u^2 + b_{003}v^3 + b_{210}3w^2u + b_{120}3w^2u + b_{021}3u^2v + b_{102}3uv^2 + b_{012}3u^2v + b_{111}6uvw \]
3.2.3 Normal control points

The normal control points are described independently from the vertex control points. As linear interpolation between the corner normals will not pick up on inflections in the curve, as shown in figure 3.5, a quadratic scheme was chosen to interpolate the normals. This results in six control points divided into two groups. $n_{200}, n_{020}$ and $n_{002}$ are the corner control points and coincide with the normals of the original triangle. $n_{110}, n_{011}$ and $n_{101}$ are placed along the border of the triangle. In order to pick up any inflections in the curve each border normal is calculated by finding the average of the corner normals belonging to that edge and reflecting the result across the plane perpendicular to the edge, as shown in figure 3.6.

Recall that the reflection $A'$ of a vector $A$ across a plane with the unnormalized normal direction $B$ is $A' = A - 2vB$, $v = (B \cdot A)/(B \cdot B)$. In this equation $\cdot$ is used to denote the dot product. Using this formula we get the following equation:

$$n_{ijk} = N_x + N_y - 2 \left( \frac{P_y - P_x}{P_y - P_x} \cdot (N_x + N_y) \right) \left( P_y - P_x \right)$$

Here $x$ and $y$ represent the first and second non-zero value among $ijk$. Note that this equation gives us an unnormalized normal.

Using these control points the normal for each point on the curve can be calculated. This formula is similar to how the vertex points are computed, albeit a quadratic scheme is used as opposed to a cubic scheme.

$$n(u, v) = \sum_{i+j+k=2} n_{ijk} \frac{2!}{i!j!k!} u^i v^j w^k$$

$$= n_{200}w^2 + n_{020}u^2 + n_{002}v^2 + n_{110}wu + n_{011}uv + n_{101}wv$$

![Figure 3.5 – Normal control points](image1)

![Figure 3.6 – Border normals are calculated by reflecting the average of the two edge normals.](image2)

![Figure 3.7 – Linear interpolated normals (left), quadratically interpolated normals (right).](image3)
3.2.4 Performance

Typically in a game you want to be able to adjust the LOD, level of detail, of the meshes as it would be wasteful to render a character that is far away from the camera using a high-resolution mesh. One way of addressing this issue is to load several different LOD versions of the mesh at start-up and during rendering choose the appropriate mesh. By using tessellation we can instead load a coarse low-resolution mesh and using subdivision create a high-resolution version on the fly using the GPU that smooth out hard edges in the low-resolution mesh.

Subdivision efficiently reduces our memory footprint both on-disk and in system and video memory as only the coarse mesh is stored, compared to storing multiple versions of each mesh. As we increase the number of vertices per mesh on the GPU we also decrease the load on the bus between the CPU and the GPU which can be a potential bottleneck for an application.

Because of the recursive nature of subdivision and the fact that the increase in detail is generated during run-time we are not limited to a few LOD versions of the mesh but can instead easily scale our mesh to contain the amount of detail we desire. It is even possible to have different amounts of detail on different areas of the mesh depending on distance from the camera, or to only increase detail in areas of interest such as those close to silhouette edges.

It is also possible to gain a performance boost by doing heavy vertex operations before applying the PN-triangle algorithm and copying the result to the new triangles instead of having to do the operation for every vertex in a high-resolution mesh.

3.2.5 Visual result

The purpose of PN-triangles is to create a visually appealing result by softening triangle creases creating a smoother silhouette and better shading results. Additional techniques such as displacement mapping can be used together with subdivision to create details such as wrinkles that cannot be stored in our coarse mesh due to the low polygon count.

One potential drawback to PN-triangles is the fact that in its basic form it cannot distinguish edges that we would like to remain sharp. Vlachos discusses two potential solutions to this issue [Vlachos01], either by adding adjacency information to the mesh or by editing mesh data in a software pre-processing step.

![Figure 3.8](image-url) – Left figure shows a mesh tessellated using PN-triangles and right shows the original mesh.
4 Prototypes

4.1 Application

For our test cases we built two prototypes, one for testing tessellation using the Geometry shader and the other for utilizing the fixed function tessellator provided by DirectX 11. To get as consistent data as possible both prototypes are built using the same application and share the same base components such as mesh initializing, logging and window framework. The graphic component is built using DirectX 11 but for the Geometry shader prototype a scenario consistent with DirectX 10 requirements is emulated. Therefore shader model 4.0 will be used for this prototype when compiling the shaders. Figure 4.1 shows a more detailed overview of the application.

Figure 4.1 – Sequence diagram showing the general flow of the application. DXEffect is an abstract class containing two subclasses for rendering using our two tessellation methods.
To verify that both prototypes generate similar visual results the application supports switching between the two draw methods during run-time. As depicted in figure 4.2 both methods generate very similar results. To further check that the same number of triangles was generated for both methods querying was used to check the number of primitives sent to the pixel shader.

![Figure 4.2 – Per pixel comparison of the resulting tessellated mesh created using the Geometry shader and the fixed function tessellator.](image)

In order to ensure that the results are not affected by multiple draw calls all meshes are instanced and stored in a single vertex array which allows us to render all meshes in one draw call. To separate each mesh a mesh ID is placed in the vertex description.

The logging system is built to have minimal implication on the test data. All data is temporarily logged to the system memory and then written to a log map at certain time intervals. After the test run is over these values are written to a file. To measure the elapsed time we used the QueryPerformanceFrequency function included in the windows.h library, allowing us the make exact time measurements. As rendering times can vary from frame to frame we opted for storing average time values in order to even out any possible spikes in our measurements.

### 4.2 Tessellator Prototype

As this prototype focuses on the new pipeline stages introduced in DirectX 11, there was only need for minor changes in the base application. To make use of the tessellator we had to specify our primitive topology as a control point patchlist, as well as specifying a subdivision level to be used by the fixed function tessellator.

#### 4.2.1 GPU Implementation

The control mesh is fed to the Vertex shader and passed on to the Hull shader. No transformations are done at this point since the mesh needs to be in local space before the algorithms are applied.

As the primitives reach the Hull shader, a few attributes must be specified before tessellation can be done. For the tests done in this thesis, a triangle is chosen as input patch to the tessellator, fractional odd as partitioning scheme and a patch constant function is defined which generates the
PN-triangle control points. In this prototype the Hull shader’s task is to forward the vertex position, i.e. primitive control point, as well as any additional vertex data.

```cpp
[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[patchconstantfunc("HS_ConstantPNTriangles")]
[outputcontrolpoints(3)]
HS_ControlPointOutput HS_PNTriangles(InputPatch<VS_Output, 3> hsin, /*.....*/)
{
    HS_ControlPointOutput hsOut = (HS_ControlPointOutput)0;
    hsOut.pos = hsin[uControlPointID].pos;
    hsOut.normal = hsin[uControlPointID].normal;
    hsOut.texC = hsin[uControlPointID].texC;
    hsOut.instance = hsin[uControlPointID].instance.x;
    return hsOut;
}
```

*Figure 4.3 – Hull shader.*

In the Domain shader the new vertex points are calculated using the barycentric coordinates, the primitive control points and the PN-triangle control points.

```cpp
/* Initialization of variables */
float4 position = calculatePositionFromUVW(triangleVP, barycentricCoords, hsConstantData.controlP);

// Compute normal from quadratic control points and barycentric coords
float4 normal = calculateNormalFromUVW(triangleN, barycentricCoords, hsConstantData.controlP);

dOut.texC = interpolateTexC(dsIn[0].texC, dsIn[1].texC, dsIn[2].texC, barycentricCoords);
dOut.normal = mul(normal, fWWorldM[instance]);
dOut.pos = mul(position, fWVPM[instance]);
dOut.posW = mul(position, fWWorldM[instance]);
```

*Figure 4.4 – Domain shader.*
4.3 Geometry shader prototype 1

In order to reach higher subdivision levels using the Geometry shader we needed to make some changes to the core application to accommodate for the fact that the amount of primitives streamed out from a Geometry shader is limited. Due to these limitations we needed to create emitter triangles in order to reach a reasonable subdivision level, forcing us to render the scene using two passes. During the first rendering pass the control mesh is fed to the GPU and instead of drawing to the screen the geometry shader outputs the emitter primitives to a buffer stored in the video memory. The number of primitives output is dependent on the subdivision level so when we change subdivision level new emitters needs to be created from the control mesh. In the second pass we use the new buffer to draw our tessellated mesh; subsequent draw calls for the mesh need only use the second pass as long as the subdivision level is unchanged.

4.3.1 GPU Implementation

4.3.1.1 First Pass:

For each triangle in the mesh a number of new sub-triangles will be created to approximate the curved PN-triangle surface. However the Geometry shader limits how many vertex points that can be streamed out during one invocation of the shader. A maximum of 1024 floating point variable can be streamed out, and with a Pixel shader input size of approximately 13 floating point variables per vertex point it gives us a maximum of 78 vertex points or 26 triangles that can be streamed out in one invocation. As this only allows us to reach subdivision level 5 we need to increase the number of triangles in the mesh by adding emitter triangles. To calculate the number of emitters needed for each control mesh primitive we first have to look at the structure of our tessellated triangle. As shown in figure 2.5 each subdivision level contains triangles that match the structure of each previous level, these inner triangles are hereon referred to as structural subdivision levels. Each emitter belongs to a certain structural subdivision level and is tasked with creating sub-triangles that make up a part of that level. Which structural subdivision level the emitter belongs to and which part of the level it is tasked with creating is stored in two attributes.
As this pass only creates emitters to be used in the next pass it would be wasteful to draw pixels to the back buffer. To avoid this we disable the Pixel shader as well as the depth and stencil tests during emitter generation.

```c
//Iterate over each subdivision level and output emitters
for(unsigned int i = tessFactor; i >= 0; i--)
{
    //One emitter per corner
    for(unsigned int j = 0; j < 3; j++)
    {
        gOut.pos = gIn[j].pos.xyz;
        gOut.normal = gIn[j].normal;
        gOut.texC = gIn[j].texC;
        gOut.tessInfo = int2(i, j);
        gOut.instance = gIn[j].instance;
        triStream.Append(gOut);
    
    /*Append second and third vertex point.*/
    /*tessinfo remains unchanged.*/
    
        triStream.RestartStrip();
    }

    //Subdivision level 1 emitter
    //One emitter since level 1 only contains a single triangle
    gOut.pos = gIn[0].pos.xyz;
    gOut.normal = gIn[0].normal;
    gOut.texC = gIn[0].texC;
    gOut.tessInfo = int2(0, 0);
    gOut.instance = gIn[0].instance;
    triStream.Append(gOut);

    /*Append second and third vertex point.*/
    /*tessinfo remains unchanged.*/
```

Figure 4.5 – Generating emitters for test case 2.

### 4.3.1.2 Second Pass:

The purpose of this pass is to generate the sub-triangles that make up our tessellated surface. Firstly we need to calculate the PN-triangle control points that will be used to locate the positions and normals on the curved surface. These control points are calculated using the same function as in the tessellator prototype ensuring that the construction of the PN-triangle algorithm will not affect our test results.

The next step is to calculate which triangles are to be generated from the current emitter, which is done using the attributes set in the previous pass. The first attribute indicates where to start binding the sub-triangles and the second indicates the structural subdivision level. The second attribute is important as the structure of our tessellated triangle allows us to reuse the formula for calculating the inner triangles by only changing their barycentric coordinates depending on the structural subdivision level.

As the barycentric coordinates for each subdivision level are constant they were pre-calculated and stored in a buffer in the video memory at application start-up. This allows us to fetch the barycentric...
coordinates from the buffer using their index values. Figure 4.6 shows how the calculations of the index values are done.

```c
int outsideTess = gin[0].tessInfo.y;
int insideTess = outsideTess - 2;
//Stores the maximum subdivision level
int tessFactor = fxTessFactor.x;

// Offset to determine which side of the triangle that is currently being generated.
int cornerOffset = gin[0].tessInfo.x;
// StartOffset is used to fetch the values for the current subdivision level from the barycentric coordinate array.
int startOffset = 0;
for(int i = 1; i < tessFactor; i++)
    for(int j = 1; j > 0; j--)
        startOffset += j * 3;

for(int i = tessFactor; i > outsideTess; i--)
    startOffset += 1 * 3;

// Determines the maximum index value for the barycentric coordinates along the outer edges.
int outMax = startOffset + outsideTess * 3;
// Determines the maximum index value for the barycentric coordinates along the inner edges.
int inMax = outMax + insideTess * 3;

// Determine start index in the barycentric coordinate array for triangle generation.
int currentOuterIndex = wrap(startOffset + cornerOffset * outsideTess - 1, outMax, startOffset);
```

Figure 4.6 – Offset calculations used to find the correct index values in the barycentric coordinate array.

Depending on which emitter approach is used sub-triangle creation is handled a little differently.

In test case 2 we create three emitters per structural subdivision level during the first pass. Each emitter is tasked with creating sub-triangles for an entire border of the structural subdivision level described by the second tessellation attribute.

Test case 3 on the other hand increases the number of emitters that are streamed out during the first pass. Each emitter is only tasked with creating two connected sub-triangles forming a quad.

The difference between the two approaches is shown in figure 4.7. Lastly the triangles are transformed to screen space and rasterized in the pixel shader. It should be noted that the vertex points were streamed out as a triangle strip and not as a triangle list, described in section 2.x, as it required less vertex points to be streamed out.

Figure 4.7 – Triangles created from an emitter. Test case 3 (blue) and test case 2 (green)
4.4 Geometry shader prototype 2

For the main part of our test concerning the Geometry shader we opted for a two pass rendering solution. However as our hypothesis was that we would receive the best results for low subdivision levels we created one case that used a one pass solution. This limits how many subdivision levels we can reach but will most likely improve performance as we can render the scene in one pass.

This allowed us to reuse the core application, only modifying the GPU implementation.

4.4.1 GPU Implementation

The control mesh is fed to the Geometry shader where each triangle is expanded to a number of sub-triangles.

The main difference compared to prototype 1 is that each triangle is tasked with creating all its sub-triangles.

As this prototype focuses on situations where decreasing the tessellation range would not be a problem we can establish further limitations might also be acceptable. Therefore we chose to create a solution designed to render the mesh at a specific subdivision level to further improve performance.

The flaw of this approach is that if we want to be able to change subdivision levels during run-time it would require switching Geometry shaders, therefore we designed an alternative solution that could handle subdivision level 1 through 5 using one Geometry shader. Both methods were tested to find out if the dynamic approach would affect the performance.

4.5 Barycentric Coordinate generation

The calculation of the barycentric coordinates were done in a separate application and stored in a text file. In order to ensure that the tessellated mesh produced by both the fixed function tessellator and the Geometry shader were as similar as possible we wanted to produce barycentric coordinates that closely matches those output by the tessellator, the result can be seen in figure 4.7.

![Tessellator Subdivision Level 5](image1)

![Geometry Shader Subdivision Level 5](image2)

Figure 4.7 – Triangle generated from the barycentric coordinates provided by the tessellator (left) and using pre-calculated barycentric coordinates (right)
For each subdivision level all barycentric coordinates along the outer triangle as well as those along the inner triangles were pre-calculated. It would have been possible to only create the barycentric coordinates for the outer triangles of each subdivision level and then offsetting these values to find the structural subdivision levels inside each outer triangle. However this would require additional calculations in the Geometry shader which made us opt for pre-calculating all coordinates despite the extra memory needed to store them. Figure 4.8 depicts how the barycentric coordinates can be calculated using repeated interpolation.

**Figure 4.8** – The border control points are interpolated based on the subdivision levels. The green and red lines show the relationship between the outer and inner triangles. Using this relationship we can interpolate the values for the corner control points of the inner triangle. The inner border control points can then be calculated in the same fashion as the outer border control points.
5 Performance testing

In this section a few different implementations for rendering a number of meshes using PN-triangles will be tested.

<table>
<thead>
<tr>
<th>Test computer specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GPU</strong></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
</tr>
<tr>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td><strong>OS</strong></td>
</tr>
<tr>
<td><strong>DirectX</strong></td>
</tr>
</tbody>
</table>

For each case four versions containing one or more meshes consisting of different amounts of polygons was tested. This was done to see how an increase in the number of vertices rendered per frame affected the performance.

![Figure 5.1](image1) ![Figure 5.2](image2)

**Figure 5.1** – 1 Cube (control mesh) consists of 12 triangles, 20 Cubes consists of 240 triangles.

**Figure 5.2** – 1 Wizard (control mesh) consists of 1986 triangles, 20 Wizards consists of 39720 triangles.

Each version was executed three times using a release build .exe file and the average result values were then calculated. During the test each subdivision level were allowed to render for five seconds to even out any potential spikes in the results.

General setup used by all cases:

- Backface culling – On
- Rendering mode – Solid
- Vsync – Off
- Depth buffer – On
- Stencil buffer – On
5.1 Test cases

5.1.1 Case 1 – Single pass
The basis for this test is to tessellate as much as possible in one draw call using the Geometry shader. This test only shows results up to subdivision level 5 as it was the maximum depth we could reach in one draw call.

Version 1 – using two static solutions designed specifically for outputting subdivision level 3 and 5.

Version 2 – using a dynamic solution allowing us to use the same shader for subdivision 1 – 5.

5.1.2 Case 2 – Focused emitting
This test uses a two pass solution, creating emitter triangles which in turn render the tessellated mesh. Each emitter outputs a border of sub-triangles, further explained in section 4.3.1.2.

5.1.3 Case 3 – Workload balancing
This test case uses the same setup as Test Case 2 with a different amount of sub-triangles generated by each emitter. In this case each emitter outputs one quad.

5.1.4 Case 4 – Stored tessellation
For this test we use the same setup as Test Case 2 except instead of rendering the mesh to the screen the resulting tessellated mesh is stored to a buffer that can be used to render the tessellated mesh with the Geometry shader disabled.

5.1.5 Case 5 – Fixed function tessellator
This test case utilizes the fixed function tessellator.

5.1.6 Case 6 – Fixed function tessellator + Geometry shader
For this test we use the same setup as Test Case 5, with the addition of a geometry shader. The sole function of the geometry shader for this test is to forward the primitives to the pixel shader.
5.2 Test scenarios

5.2.1 Scenario 1 – One cube

Figure 5.3 – Scenario 1.

5.2.2 Scenario 2 – 20 Cubes

Figure 5.4 – Scenario 2.
5.2.3 Scenario 3 – One Wizard

![Figure 5.5 – Scenario 3]

5.2.4 Scenario 4 – 20 Wizards

![Figure 5.6 – Scenario 4]
5.3 Test case conclusions

5.3.1 Test Case 1 Conclusions:
This test case performed fairly well for low subdivision levels. As can be seen in the graph the static implementation excelled at subdivision level 3 but was then taken over by the looped implementation. The static implementation probably performed best at first because some of the loops in the other test version will only execute for one iteration. This shows that tailoring the Geometry shader to the specific needs of the application will have a noticeable effect on performance results. The main drawback to a single pass implementation is that we will only be able to output low subdivision levels as the number of floating point variables that can be streamed out from the Geometry shader is limited.

5.3.2 Test Case 2 Conclusions:
Since this was a more general solution to solving tessellated rendering it allowed us to reach higher subdivision levels than test case 1. However looking at the results we can see that striving for a general solution decreased the performance results. Looking at the graph we can see that the effect on rendering time increased by roughly 130% for each subdivision level across all four test runs, giving us a good indication on how this implementation would perform for varying number of meshes.

5.3.3 Test Case 3 Conclusions:
Increasing the number of emitters turned out to decrease the performance of the application as can be seen in the graph. Contrary to our belief that this would balance the workload between creating emitters and rendering sub-triangles the increased number of emitters created more of a bottleneck than having each emitter create a border of sub-triangles.

Figure 5.7 - Memory consumption rendering 20 Wizards. The values where gathered by multiplying the number of vertices in the buffers with the size of each vertex point, and double checked using PIX to control buffer size.
5.3.4 Test Case 4 Conclusions:
Considering only the performance results test case 4 was able to produce the most number of frames per second at the higher end of our test spectrum. However the drawback to this solution is the video memory consumption as can be seen in figure 5.7. A mesh rendered at subdivision level 11 using stored tessellation will require roughly 200 times the amount of memory compared to tessellating the mesh without storing the result.

5.3.5 Test Case 5 Conclusions:
Using the fixed function tessellator to render the meshes gave us a rendering time well below any of the Geometry shader implementations. Looking at the rendering time for subdivision level 1, basically the control mesh, we see that the tessellator adds some overhead increasing the rendering time compared to rendering the mesh without applying any tessellation algorithm. To see how the overhead affects rendering time we can look at test case 4 as it stores the mesh and then renders it using only the Vertex- and Pixel shader which is how a non-tessellated mesh would typically be rendered. However the tessellator is able to switch between rendering the mesh at subdivision level 1 and 3 without seeing a decrease in performance.

5.3.6 Test Case 6 Conclusions:
As can be seen in test case 6, attaching a Geometry shader after the fixed function tessellator will heavily affect the performance. In this study no real work was done in this Geometry shader but adding more calculations would potentially affect the performance even more. Although compared to the results of the Geometry shader implementations, the overall performance using the tessellator would probably still be better.

As can be seen in the figure 5.6 the fixed function tessellator outperformed nearly all Geometry shader implementations for all mesh variants on every subdivision levels. As predicted in our hypothesis the Geometry shader results were closest to that of the tessellator at low subdivision levels. However even the static subdivision level 3 Geometry shader was only able to output frames at a rate of one third the speed of the tessellator.
6 Conclusions and further work

6.1.1 Discussion

The focus of this thesis was to compare the performance of rendering a tessellated mesh using the Geometry shader as opposed to using the fixed function tessellator provided by DirectX 11. Several different approaches to implementing tessellation using the Geometry shader were done in order to find the best solution. Due to limitations in the amount of primitives that the Geometry shader is able to stream out we had to split our solutions into two groups. One focused on a more general approach allowing us to reach higher subdivision levels and the other focusing on the performance when rendering meshes at low subdivision levels. To capitalize on the fact that the Geometry shader is able to stream out vertices to a buffer, we designed a solution that streamed out a tessellated mesh which could then be rendered without redoing any tessellation calculations. The positive aspect of this approach compared to traditional LOD rendering is that we only need to make the buffer large enough to accommodate the maximum subdivision level to be rendered, instead of one buffer for each LOD version of the mesh.

Looking at the test results we can see that the tessellator is able to handle a wide range of subdivision levels very well. As the graph progresses we can see that the gap between using the Geometry shader and the fixed function tessellator increases.

If one was mainly concerned with smoothening silhouettes and creases then the lower end of the subdivision spectrum is the most interesting as this is where the visual appeal of the mesh changes most drastically. Looking at the test results we can see that test cases 1 and 4 would both be good candidates under this condition when considering a Geometry shader solution. The choice mainly comes down to whether memory space can be sacrificed in order to improve the performance. Comparing the two cases we see that case 4 performs roughly 7 times faster than case 1 when rendering 20 low-resolution meshes, although it requires 45 times more memory space.

When using tessellation in combination with other rendering techniques such as displacement mapping higher subdivision levels will give a more detailed representation of the displacement map. It should be noted that in order to generate a water-tight displaced surface a few preemptive measures has to be taken, detailed in [Castaño09], [Tatarchuck09]. The viable Geometry shader candidates for this scenario are test cases 2 and 4, however when rendering higher subdivision levels case 4 will require a significant amount of video memory as shown in figure 5.7.

*Figure 6.1 – An example of displacement mapping. © Nvidia, [GPUgems2]*)
It should be noted that the performance results in the tests done in this study do not indicate the performance that can potentially be achieved in a real scenario as those implementations would most likely utilize different optimization techniques such as adaptive tessellation. However in order to get comparable results all implementations had to conform to using the same algorithm.

6.1.2 Conclusions

Our hypothesis was that the Geometry shader would perform better than the tessellator for low subdivision levels as the tessellator adds some overhead to the rendering time. Looking at the results we can see that even though some overhead was added it was significantly less than appending triangles using the Geometry shader, disproving our hypothesis.

The main drawback to rendering a tessellated mesh using the Geometry shader is the fact that in order to reach higher subdivision levels a two pass rendering solution is needed, as the amount of vertices that can be output in one invocation of the Geometry shader is limited. It is possible to implement a solution for rendering tessellated meshes using the Geometry shader that will perform at acceptable speeds, although it is not practical to choose this method if the possibility exists to use the fixed function tessellator.

The tessellator also has a few advantages over a Geometry shader approach. As the results from the fact that the Hull Shader and patch constant function are available in each invocation of the Domain shader belonging to the same patch, heavy calculations such as calculating the control points need only be done once for each patch. A Geometry shader approach would have to split the patch into smaller parts forcing these calculations to be redone for each part of the patch. Looking at query information detailing GPU pipeline statistics it is clear that the tessellator can create sub-triangles using very few vertex points. For example subdivision level 5 divides each triangle into 37 sub-triangles, which using the tessellator can be created from 37 invocations of the Domain shader each outputting one vertex point.

Figure 6.1 – High-resolution mesh generated using smooth tool in Maya (left). Tessellated mesh generated at subdivision level 3 using the Geometry shader (middle) and the fixed function tessellator (right).
6.1.3 Further work

Considering the test results the most interesting subject for further work is optimizing a Geometry shader approach that focuses on rendering tessellated meshes using low subdivision levels. In this thesis we concentrated on creating the same visual results as the tessellator, without this requirement there can be room for changing how the coordinates are spread out over the control primitives potentially allowing for better performance.

Another interesting aspect is that as the triangles in the control mesh are actually never rendered but instead only used as a basis for creating sub-triangles, it may be possible to improve how they are stored. One possibility is to store the mesh as a point-list, where every point contains the information needed to describe a triangle. This could potentially avoid stalling in the pipeline as the Geometry shader has to wait for a complete primitive to arrive. The fact that switching from using triangle lists to triangle strips in the Geometry shader prototype significantly increased the performance shows the value of efficient pipeline usage.
7 References

7.1 Bibliography


7.2 Websites


7.3 Images

8 Appendix A – Tessellator prototype

High Level Shader Language source code used in the tessellator prototype.

```hs
HS_ConstantOutput HS_ConstantPNTriangles(InputPatch<VS_Output, 3> hsIn)
{
    HS_ConstantOutput hsOut = (HS_ConstantOutput)0;

    // Set the tessellation-factors, currently the same for edge and inside
    hsOut.tessFactor[0] = fxTessFactor.x;
    hsOut.tessFactor[1] = fxTessFactor.x;
    hsOut.tessFactor[2] = fxTessFactor.x;
    hsOut.insideTessFactor = fxTessFactor.x;

    float4 triangleVP[3];
    triangleVP[0] = hsIn[0].pos;
    triangleVP[1] = hsIn[1].pos;

    float3 triangleN[3];
    triangleN[0] = hsIn[0].normal;
    triangleN[1] = hsIn[1].normal;

    hsOut.controlP = calculateControlPoints(triangleVP, triangleN);

    return hsOut;
}

[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[patchconstantfunc("HS_ConstantPNTriangles")]
[outputcontrolpoints(3)]
HS_ControlPointOutput HS_PNTriangles(InputPatch<VS_Output, 3> hsIn, uint uControlPointID : SV_OutputControlPointID)
{
    HS_ControlPointOutput hsOut = (HS_ControlPointOutput)0;

    hsOut.pos = hsIn[uControlPointID].pos;
    hsOut.normal = hsIn[uControlPointID].normal;
    hsOut.texC = hsIn[uControlPointID].texC;
    hsOut.instance = hsIn[uControlPointID].instance.x;

    return hsOut;
}
```


PS_Input DS_PNTriangles(HS_ConstantOutput hsConstantData, const OutputPatch<HS_ControlPointOutput, 3> dsIn, float3 barycentricCoords : SV_DomainLocation)
{
    PS_Input dsOut = (PS_Input)0;
    int instance = dsIn[0].instance;

    float4 triangleVP[3];
    triangleVP[0] = float4(dsIn[0].pos, 1.0f);
    triangleVP[1] = float4(dsIn[1].pos, 1.0f);
    triangleVP[2] = float4(dsIn[2].pos, 1.0f);

    float3 triangleN[3];
    triangleN[0] = dsIn[0].normal;
    triangleN[1] = dsIn[1].normal;

    float4 position = calculatePositionfromUVW(triangleVP, barycentricCoords, hsConstantData.controlP);

    // Compute normal from quadratic control points and barycentric coords
    float4 normal = calculateNormalfromUVW(triangleN, barycentricCoords, hsConstantData.controlP);

    dsOut.texC = interpolateTexC(dsIn[0].texC, dsIn[1].texC, dsIn[2].texC, barycentricCoords);
    dsOut.normal = mul(normal, fxWorldM[instance]);
    dsOut.pos = mul(position, fxWVPM[instance]);
    dsOut.posW = mul(position, fxWorldM[instance]);

    return dsOut;
}
9 Appendix B – Geometry shader prototype

High Level Shader Language source code used by case 1.2 and case 2.

9.1 Appendix B1 – Case 1.2

/** Generates PN-triangles
   Takes an emitter as argument and generates subdivision triangles for a triangle edge.
   TessInfo tells us which edge should be generated and at what subdivision level.
 */
[maxvertexcount(65)]
void GS_PNTriangles(triangle VS_Output gIn[3], inout TriangleStream<PS_Input> triStream)
{
    //Calculate control points
    float4 triangleVP[3];
    triangleVP[0] = gIn[0].pos;
    triangleVP[1] = gIn[1].pos;
    triangleVP[2] = gIn[2].pos;

    float3 triangleN[3];
    triangleN[0] = gIn[0].normal;
    triangleN[1] = gIn[1].normal;
    triangleN[2] = gIn[2].normal;
    ControlPoints controlP = (ControlPoints)0;
    controlP = calculateControlPoints(triangleVP, triangleN);

    int instance = gIn[0].instance;

    //Save the outsideTess (current subdivision level) & insideTess(previous subdivision level)
    //OutsideTess is used for the subdivision level currently being generated & insideTess is used to access bind points
    //on the inner triangle

    //Stores the maximum subdivision level
    int tessFactor = fxTessFactor.x;

    //Offset to determine which side of the triangle that is currently being generated.
    int cornerOffset = 0;
    //Startoffset is used to fetch the values for the current subdivision level from the barycentric coordinate array.
    int startOffset = 0;
    for(int i = 1; i < tessFactor; i+=2)
        for(int j = i; j >= 1; j -= 2)
            startOffset += j * 3;

    PS_Input gOut = (PS_Input)0;
    float4 PNpos = float4(0.0f, 0.0f, 0.0f, 0.0f);
    float4 PNnormal = float4(0.0f, 0.0f, 0.0f, 0.0f);

    //Indices for constructing the triangles inbetween corners
    //Wrap to make sure it stays inside the index boundaries of the inner triangle
    int oindexA = 0;
    int oindexB = 0;
    int oindexC = 0;

    //Indices for constructing corner triangles
    int iindexA = 0;
    int iindexB = 0;
    int iindexC = 0;

    int outsideTess = 0;
    int insideTess = 0;
    int outMax = 0;
    int inMax = 0;
for(int i = tessFactor; i > 1; i /= 2) {
    outsideTess = i;
    insideTess = outsideTess - 2;

    // Determines the maximum index value for the barycentric coordinates along the outer edges.
    outMax = startOffset + outsideTess * 3;
    // Determines the maximum index value for the barycentric coordinates along the inner edges.
    inMax = outMax + insideTess * 3;

    for(int j = 0; j < 3; j++) {
        cornerOffset = j;
        // Determine start index in the barycentric coordinate array for triangle generation.
        int currentOuterIndex = wrap(startOffset + cornerOffset * outsideTess, outMax, startOffset);

        // First corner triangle
        indexA = wrap(outMax + (cornerOffset * insideTess), inMax, outMax);
        indexB = currentOuterIndex;
        indexC = wrap(indexB + 1, outMax, startOffset);

        PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[indexB], controlP);
        PNNormal = calculateNormalFromUVW(triangleN, barycentricUVW[indexB], controlP);
        gOut.pos = mul(PNpos, fxWVP[instance]);
        gOut.posW = mul(PNpos, fxWorldM[instance]);
        gOut.normal = mul(PNNormal, fxWorldM[instance]);
        gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[indexB]);
        triStream.Append(gOut);

        PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[indexB], controlP);
        PNNormal = calculateNormalFromUVW(triangleN, barycentricUVW[indexB], controlP);
        gOut.pos = mul(PNpos, fxWVP[instance]);
        gOut.posW = mul(PNpos, fxWorldM[instance]);
        gOut.normal = mul(PNNormal, fxWorldM[instance]);
        gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[indexB]);
        triStream.Append(gOut);

        PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[indexA], controlP);
        PNNormal = calculateNormalFromUVW(triangleN, barycentricUVW[indexA], controlP);
        gOut.pos = mul(PNpos, fxWVP[instance]);
        gOut.posW = mul(PNpos, fxWorldM[instance]);
        gOut.normal = mul(PNNormal, fxWorldM[instance]);
        gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[indexA]);
        triStream.Append(gOut);

        currentOuterIndex = wrap(currentOuterIndex + 1, outMax, startOffset);
        indexB = currentOuterIndex;
        currentOuterIndex = currentOuterIndex + 1;
        indexC = currentOuterIndex;

        PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[indexC], controlP);
        PNNormal = calculateNormalFromUVW(triangleN, barycentricUVW[indexC], controlP);
        gOut.pos = mul(PNpos, fxWVP[instance]);
        gOut.posW = mul(PNpos, fxWorldM[instance]);
        gOut.normal = mul(PNNormal, fxWorldM[instance]);
        gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[indexC]);
        triStream.Append(gOut);

        oindexA = indexA;
// Iterate over coordinates along the edge
for(int k = 0; k < insideTess; k++)
{
    currentOuterIndex = wrap(currentOuterIndex + 1, outMax, startOffset);

    oindexA = wrap(oindexA + 1, inMax, outMax);
    oindexB = currentOuterIndex;
    oindexC = currentOuterIndex;

    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[oindexA], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[oindexA], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[oindexA]);
    triStream.Append(gOut);

    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[oindexB], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[oindexB], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[oindexB]);
    triStream.Append(gOut);
}

triStream.RestartStrip();

startOffset += i * 3;
}

// Special case since the subdivision level 1 only generates one new triangle
iindexA = startOffset;
iindexB = startOffset + 1;
iindexC = startOffset + 2;
PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexA], controlP);
PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexA], controlP);
gOut.pos = mul(PNpos, fxWVPM[instance]);
gOut.posW = mul(PNpos, fxWorldM[instance]);
gOut.normal = mul(PNnormal, fxWorldM[instance]);
gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexA]);
triStream.Append(gOut);

PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexB], controlP);
PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexB], controlP);
gOut.pos = mul(PNpos, fxWVPM[instance]);
gOut.posW = mul(PNpos, fxWorldM[instance]);
gOut.normal = mul(PNnormal, fxWorldM[instance]);
gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexB]);
triStream.Append(gOut);

PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexC], controlP);
PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexC], controlP);
gOut.pos = mul(PNpos, fxWVPM[instance]);
gOut.posW = mul(PNpos, fxWorldM[instance]);
gOut.normal = mul(PNnormal, fxWorldM[instance]);
gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexC]);
triStream.Append(gOut);
9.2 Appendix B2 – Case 2
Application source code to render a tessellated mesh

```cpp
void DXGeometry::generateLoD(InstancedModels * pModels )
{
    // Assign stream out shader
    DXUtil::mpDeviceContext->GSSetShader(mpGeometryEmitterShader, NULL, 0);
    updateCBuffers(pModels->getMaterials(), pModels->getNrOfModels());

    // Render to the buffer
    Mesh * mesh = pModels->getMesh();
    const VertexBuffer * vb = mesh->getVertexBuffers()[0];
    const IndexBuffer * ib = mesh->getIndexBuffers()[0];

    ID3D11Buffer* vertexBuffer = vb->getBuffer();
    ID3D11Buffer* indexBuffer = ib->getBuffer();

    unsigned int stride = vb->getStride();
    unsigned int offset = 0;
    DXUtil::mpDeviceContext->IASetVertexBuffers(0, 1, &vertexBuffer, &stride, &offset );
    DXUtil::mpDeviceContext->IASetIndexBuffer(indexBuffer, DXGI_FORMAT_R32_UINT, 0);

    // Set stream out buffer as target
    ID3D11Buffer * vertexBufferSO = mesh->getVertexBuffers()[1]->getBuffer();
    unsigned int offsetSO = 0;
    DXUtil::mpDeviceContext->SOSetTargets(1,&vertexBufferSO,&offsetSO);

    // Disable PS and depth stencil buffer to avoid unnecessary rendering
    ID3D11PixelShader * unbindPS = NULL;
    DXUtil::mpDeviceContext->PSSetShader(unbindPS,NULL,0);
    DXUtil::mpDeviceContext->OMSetDepthStencilState(mpDepthStencilStates[DSTATE_DISABLE], 0);

    DXUtil::mpDeviceContext->DrawIndexed(ib->getNrOfIndices(), 0, 0);
    LoDGenerated = true;
    mesh->swapBuffer(1, 1, 2);

    // Unbind the buffers.
    ID3D11Buffer * unbindSo[1] = {0};
    DXUtil::mpDeviceContext->SOSetTargets(1,unbindSo,&offset);
    ID3D11Buffer * unbindV[1] = {0};
    DXUtil::mpDeviceContext->IASetVertexBuffers(0,1,unbindV,&stride,&offset);

    // Reset to previous shaders and settings
    DXUtil::mpDeviceContext->PSSetShader(mpPixelShader,NULL,0);
    DXUtil::mpDeviceContext->OMSetDepthStencilState(mpDepthStencilStates[DSTATE_ENABLE], 0);
    DXUtil::mpDeviceContext->GSSetShader(mpGeometryShader, NULL, 0);
}
```

```cpp
void DXGeometry::renderMesh( Mesh* mesh )
{
    if(!LoDGenerated)
        DX11Effect::renderMesh(mesh);
    else
    {
        DXUtil::mpDeviceContext->VSSetShader(mpVertexShaderGSTess,NULL, 0);

        const VertexBuffer * vb = mesh->getVertexBuffers()[2];

        ID3D11Buffer* vertexBuffer = vb->getBuffer();
        unsigned int stride = vb->getStride();
```
unsigned int offset = 0;

DXUtil::mpDeviceContext->IASetInputLayout(mpInputLayoutGSTess);
DXUtil::mpDeviceContext->IASetVertexBuffers(0, 1, &vertexBuffer, &stride, &offset);
DXUtil::mpDeviceContext->DrawAuto();
}

High Level Shader Language source code.

/** Streams emitter triangles to a buffer
   For each triangle in the mesh, new emitter triangles are streamed out to a buffer.
   The number of emitters depends on subdivision level
*/
[maxvertexcount(65)]
void GS_CalculateEmitterTriangles(triangle VS_Output gIn[3], inout TriangleStream<VS_GSTess_Input> triStream) {
    VS_GSTess_Input gOut = (VS_GSTess_Input)0;
    //Stores the maximum subdivision level
    int tessFactor = fxTessFactor.x;
    //Iterate over each subdivision level and output emitters
    for(unsigned int i = tessFactor; i >= 3; i-=2) {
        //One emitter per corner
        for(unsigned int j = 0; j < 3; j++) {
            gOut.pos = gIn[j].pos.xyz;
            gOut.normal = gIn[j].normal;
            gOut.texC = gIn[j].texC;
            gOut.tessInfo = int2(j, i);
            gOut.instance = gIn[j].instance;
            triStream.Append(gOut);
        }
        //Subdivision level 1 emitter
        //One emitter since level 1 only contains a single triangle
        gOut.pos = gIn[0].pos.xyz;
        gOut.normal = gIn[0].normal;
        gOut.texC = gIn[0].texC;
        gOut.tessInfo = int2(0, 1);
        gOut.instance = gIn[0].instance;
        triStream.Append(gOut);
    }
}
```cpp
triStream.RestartStrip();
}

/**
 * Generates PN-triangles
 * Takes an emitter as argument and generates subdivision triangles for a triangle edge.
 * TessInfo tells us which edge should be generated and at what subdivision level.
 */
[maxvertexcount(65)]
void GS_PNTriangles(triangle VS_GSTess_Output gIn[3], inout TriangleStream<PS_Input> triStream)
{
    //Calculate control points
    float4 triangleVP[3];
    triangleVP[0] = gIn[0].pos;
    triangleVP[1] = gIn[1].pos;
    triangleVP[2] = gIn[2].pos;

    float3 triangleN[3];
    triangleN[0] = gIn[0].normal;
    triangleN[1] = gIn[1].normal;
    triangleN[2] = gIn[2].normal;
    ControlPoints controlP = (ControlPoints)0;
    controlP = calculateControlPoints(triangleVP, triangleN);

    int instance = gIn[0].instance;

    //Save the outsideTess (current subdivision level) & insideTess(previous subdivision level)
    //OutsideTess is used for the subdivision level currently being generated & insideTess is used to access bind points
    //on the inner triangle
    int outsideTess = gIn[0].tessInfo.y;
    int insideTess = outsideTess - 2;

    //Stores the maximum subdivision level
    int tessFactor = fxTessFactor.x;

    //Offset to determine which side of the triangle that is currently being generated.
    int cornerOffset = gIn[0].tessInfo.x;

    //Startoffset is used to fetch the values for the current subdivision level from the barycentric coordinate array.
    int startOffset = 0;
    for(int i = 1; i < tessFactor; i+=2)
        for(int j = i; j >= 1; j-=2)
            startOffset += j * 3;

    for(int i = tessFactor; i > outsideTess; i-=2)
        startOffset += i * 3;

    //Determines the maximum index value for the barycentric coordinates along the outer edges.
    int outMax = startOffset + outsideTess * 3;

    //Determines the maximum index value for the barycentric coordinates along the inner edges.
    int inMax = outMax + insideTess * 3;

    //Determine start index in the barycentric coordinate array for triangle generation.
    int currentOuterIndex = wrap(startOffset + cornerOffset * outsideTess - 1, outMax, startOffset);

    PS_Input gOut = (PS_Input)0;
    float4 PNpos = float4(0.0f, 0.0f, 0.0f, 0.0f);
    float4 PNnormal = float4(0.0f, 0.0f, 0.0f, 0.0f);
```
// Indices for constructing the triangles in-between corners
// Wrap to make sure it stays inside the index boundaries of the inner triangle
int oindexA = wrap(outMax + (cornerOffset*insideTess), inMax, outMax);
int oindexB = 0;
int oindexC = 0;

// Indices for constructing corner triangles
int iindexA = 0;
int iindexB = 0;
int iindexC = 0;

// Special case since the subdivision level 1 only generates one new triangle
if(outsideTess == 1)
{
    iindexA = startOffset;
    iindexB = startOffset+1;
    iindexC = startOffset+2;
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexA], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexA], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexA]);
    triStream.Append(gOut);
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexB], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexB], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexB]);
    triStream.Append(gOut);
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexC], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexC], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexC]);
    triStream.Append(gOut);
}
else
{
    // First corner triangle
    iindexA = oindexA;
    iindexB = currentOuterIndex;
    iindexC = wrap(indexB + 1, outMax, startOffset);
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexB], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexB], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexB]);
    triStream.Append(gOut);
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexC], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexC], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexC]);
    triStream.Append(gOut);
    PNpos = calculatePositionFromUVW(triangleVP, barycentricUVW[iindexD], controlP);
    PNnormal = calculateNormalFromUVW(triangleN, barycentricUVW[iindexD], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexD]);
    triStream.Append(gOut);
}

PNpos = calculatePositionfromUVW(triangleVP, barycentricUVW[iindexA], controlP);
PNnormal = calculateNormalfromUVW(triangleN, barycentricUVW[iindexA], controlP);
gOut.pos = mul(PNpos, fxWVPM[instance]);
gOut.posW = mul(PNpos, fxWorldM[instance]);
gOut.normal = mul(PNnormal, fxWorldM[instance]);
gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexA]);
triStream.Append(gOut);

//Second corner triangle
currentOuterIndex = wrap(currentOuterIndex + 1, outMax, startOffset);
iindexB = currentOuterIndex;
currentOuterIndex = currentOuterIndex + 1;
iindexC = currentOuterIndex;

PNpos = calculatePositionfromUVW(triangleVP, barycentricUVW[iindexC], controlP);
PNnormal = calculateNormalfromUVW(triangleN, barycentricUVW[iindexC], controlP);
gOut.pos = mul(PNpos, fxWVPM[instance]);
gOut.posW = mul(PNpos, fxWorldM[instance]);
gOut.normal = mul(PNnormal, fxWorldM[instance]);
gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[iindexC]);
triStream.Append(gOut);

//Iterate over coordinates along the edge
for(int j = 0; j < insideTess; j++)
{
    currentOuterIndex = wrap(currentOuterIndex + 1, outMax, startOffset);
    oindexA = wrap(oindexA + 1, inMax, outMax);
    oindexB = currentOuterIndex;
    oindexC = currentOuterIndex;

    PNpos = calculatePositionfromUVW(triangleVP, barycentricUVW[oindexA], controlP);
    PNnormal = calculateNormalfromUVW(triangleN, barycentricUVW[oindexA], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[oindexA]);
    triStream.Append(gOut);
    PNpos = calculatePositionfromUVW(triangleVP, barycentricUVW[oindexB], controlP);
    PNnormal = calculateNormalfromUVW(triangleN, barycentricUVW[oindexB], controlP);
    gOut.pos = mul(PNpos, fxWVPM[instance]);
    gOut.posW = mul(PNpos, fxWorldM[instance]);
    gOut.normal = mul(PNnormal, fxWorldM[instance]);
    gOut.texC = interpolateTexC(gIn[0].texC, gIn[1].texC, gIn[2].texC, barycentricUVW[oindexB]);
    triStream.Append(gOut);
}
}
**10 Appendix C – Helper functions**

```cpp
float4 calculatePositionfromUVW(float4 triangleVP[3], float3 barycentricCoords, ControlPoints controlP) {
    // The barycentric coordinates
    float u = barycentricCoords.x;
    float v = barycentricCoords.y;
    float w = barycentricCoords.z;

    // Precompute squares and squares * 3
    float uu = u * u;
    float vv = v * v;
    float ww = w * w;

    float uu3 = uu * 3.0f;
    float vv3 = vv * 3.0f;
    float ww3 = ww * 3.0f;

    // Compute position from cubic control points and barycentric coords
    float3 position = triangleVP[0] * ww * w +
                     triangleVP[1] * uu * u +
                     triangleVP[2] * vv * v +
                     controlP.B210 * ww3 * u +
                     controlP.B120 * w * uu3 +
                     controlP.B201 * ww3 * v +
                     controlP.B021 * uu3 * v +
                     controlP.B102 * w * vv3 +
                     controlP.B012 * u * vv3 +
                     controlP.B111 * 6.0f * w * u * v;

    return float4(position, 1.0f);
}

float4 calculateNormalfromUVW(float3 triangleN[3], float3 barycentricCoords, ControlPoints controlP) {
    // The barycentric coordinates
    float u = barycentricCoords.x;
    float v = barycentricCoords.y;
    float w = barycentricCoords.z;

    // Precompute squares and squares * 3
    float uu = u * u;
    float vv = v * v;
    float ww = w * w;

    float3 normal = triangleN[0] * ww +
                   triangleN[1] * uu +
                   triangleN[2] * vv +
                   controlP.N110 * w * u +
                   controlP.N011 * u * v +
                   controlP.N101 * w * v;

    normal = normalize(normal);

    return float4(normal, 0.0f);
}

float2 interpolateTexC(float2 tex1, float2 tex2, float2 tex3, float3 barycentricCoord) {
    return tex2 * barycentricCoord.x + tex3 * barycentricCoord.y + tex1 * barycentricCoord.z;
}
```
/** Calculate control points */

ControlPoints calculateControlPoints(float4 pos[3], float3 normal[3])
{
    ControlPoints retval = (ControlPoints)0;

    // Set the positions
    float3 B300 = pos[0]; // P3
    float3 B030 = pos[1]; // P2
    float3 B003 = pos[2]; // P1

    // Set the normals
    float3 N200 = normal[0]; // N3
    float3 N020 = normal[1]; // N2
    float3 N002 = normal[2]; // N1

    // Spread the points over the "surface"
    retval.B210 = (2 * B300 + B030 - dot((B030 - B300), N200) * N200) / 3;
    retval.B120 = (2 * B030 + B300 - dot((B300 - B030), N020) * N020) / 3;
    retval.B021 = (2 * B030 + B003 - dot((B003 - B030), N020) * N020) / 3;
    retval.B012 = (2 * B003 + B030 - dot((B030 - B003), N002) * N002) / 3;
    retval.B102 = (2 * B003 + B300 - dot((B300 - B003), N002) * N002) / 3;
    retval.B201 = (2 * B300 + B003 - dot((B003 - B300), N200) * N200) / 3;

    // Calculate the center point
    float3 V = (B003 + B030 + B300) / 3;
    retval.B111 = E + (E - V)/2;

    // Reflect end-normal perpendicular to the plane
    float v12 = 2.0f * dot((B030 - B300), (N200 + N020)) / dot((B030 - B300), (B030 - B300));
    float v23 = 2.0f * dot((B003 - B030), (N020 + N002)) / dot((B003 - B030), (B003 - B030));
    float v31 = 2.0f * dot((B300 - B003), (N002 + N200)) / dot((B300 - B003), (B300 - B003));

    retval.N110 = normalize(N200 + N020 - v12 * (B030 - B300));
    retval.N011 = normalize(N020 + N002 - v23 * (B003 - B030));
    retval.N101 = normalize(N002 + N200 - v31 * (B300 - B003));

    return retval;
}

/** wraps value to a specific range */

int wrap(int inval, int max, int min)
{
    if(inval < min)
        return max - (min - inval);
    else if(inval > max)
        return min + (inval - max);
    else
        return inval;
}