Thesis no: BCS-2016-02

Silhouette-based Level of Detail

A comparison of real-time performance and image space metrics

Jonas Andersson

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author:
Jonas Andersson
E-mail: joac13@student.bth.se

University advisor:
M. Sc. Francisco Lopez Luro
Dept. of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Context. The geometric complexity of objects in games and other real-time applications is a crucial aspect concerning the performance of the application. Such applications usually redraw the screen between 30-60 times per second, sometimes even more often, which can be a hard task in an environment with a high number of geometrically complex objects. The concept called Level of Detail, often abbreviated LoD, aims to alleviate the load on the hardware by introducing methods and techniques to minimize the amount of geometry while still maintaining the same, or very similar result.

Objectives. This study will compare four of the often used techniques, namely Static LoD, Unpopping LoD, Curved PN Triangles, and Phong Tessellation. Phong Tessellation is silhouette-based, and since the silhouette is considered one of the most important properties, the main aim is to determine how it performs compared to the other three techniques.

Methods. The four techniques are implemented in a real-time application using the modern rendering API Direct3D 11. Data will be gathered from this application to use in several experiments in the context of both performance and image space metrics.

Conclusions. This study has shown that all of the techniques used works in real-time, but with varying results. From the experiments it can be concluded that the best technique to use is Unpopping LoD. It has good performance and provides a good visual result with the least average amount of popping of the compared techniques. The dynamic techniques are not suitable as a substitute to Unpopping LoD, but further research could be conducted to examine how they can be used together, and how the objects themselves can be designed with the dynamic techniques in mind.

Keywords: real-time, level of detail, tessellation

CONTENTS

1 INTRODUCTION

The geometric complexity of objects in games and other real-time applications is a crucial aspect concerning the performance of the application. Such applications usually redraw the screen between 30-60 frames per second (FPS), sometimes even more often. If the hardware had unlimited power and resources this would not be an issue, the objects could be of unlimited detail. However, this is not the case, and redrawing an environment with a high number of geometrically complex objects without missing the deadline of ~16.67 milliseconds (60 FPS) can be a hard task.

Often games aim to achieve graphics as realistic as possible, and the geometric complexity is one of the big factors. If an object has more geometry it will most likely be able to resemble the real world more closely. That is why developers aim to have as high quality as possible, while still having an application that will reach the performance goals.

Objects in a game consist of a number of primitives, most often triangles. Together these triangles form the model, often referred to as the mesh of the object, which is the visual representation in three dimensions. When the object is rendered it is projected from three dimensions to the two dimensional space of the screen. Many of the triangles will then be invisible due to, for example, their projection on the final image being less than the pixel resolution of the screen, or because their normal is facing away from the camera. The resources spent on handling these triangles in the graphics pipeline is therefore wasted. Instead the aim is to spend the resources where it creates the most noticeable difference, for example, on objects closer to the screen.

The concept called Level of Detail [2], often abbreviated LoD, aims to alleviate the load on the graphics pipeline by introducing methods and techniques to minimize the geometric complexity by only using the level of detail that is necessary to maintain the same visual result.

This study will provide a comparison of four of these techniques. The comparison will be in two parts. The first part will compare the performance in FPS and number of triangles. The second part will provide a comparison of the rendered images of the different techniques.

1.1 Background and Related Work

The field of Level of Detail can be traced back to an article by James H. Clark in 1976 [1] in which he proposed an idea of objects in computer graphics having varying amount of detail depending on the distance from the screen:

> "For example, a dodecahedron looks like a sphere from a sufficiently large distance and thus can be used to model it so long as it is viewed from that or a greater distance. However, if it must ever be viewed more closely, it will look like a dodecahedron. One solution to this is simply to define it with the most detail that will ever be necessary. However, then it might have far more detail than is needed to represent it at large distances, and in a complex environment with many such objects, there would be too many polygons (or other geometric primitives) for the visible surface algorithms to efficiently handle." [1, p. 550-551]

The field has since been widely researched, and many different methods trying to accomplish this has been developed. Many of them focus on real-time applications, while some of them are used for other applications. Since games are rendered in real-time, the techniques relevant will also have to be adapted to it.

Most of the techniques need the object to have several different models with varying amount of detail, from low to high, as can be seen in (a) in Figure 1 below.

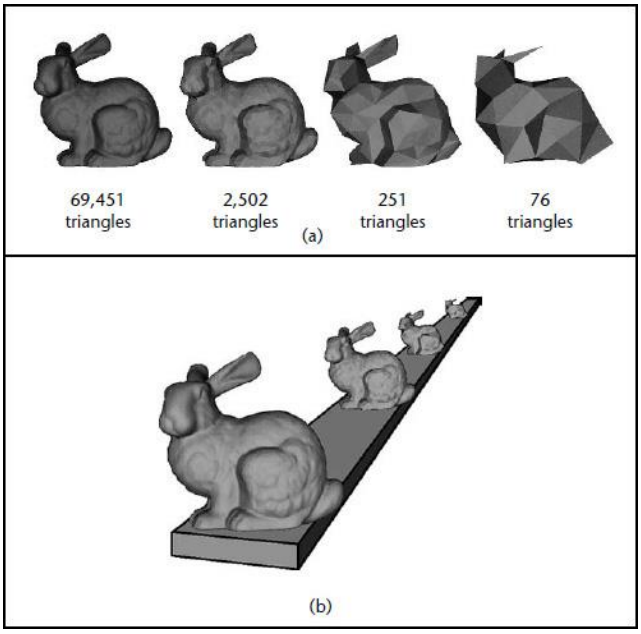

Figure 1. (a) Example of how a complex object is simplified. (b) Example showing how the models can be used depending on distance criteria. [2, p. 5].

One of the techniques is called Static LoD, which is one of the simplest ones. Based on some criteria, such as the distance from the camera or the area of the screen occupied, it will simply use the model with the appropriate level of detail. An example of this can be seen in (b) in Figure 1 above, where distance is the criteria.

The problem with Static LoD is that it suffers from a problem with temporal visual discontinuations called *popping* which occurs when switching between the models. The switch from a low quality model to a high quality model, or vice versa, often creates a visible *pop*.

A technique that tries to alleviate the problem with popping is called Unpopping LoD [3]. It does so with the help of the blending functionality of the graphics hardware. During an interval of, for example, time or distance, the technique renders both the current model and the new model at the same time, blending them together in image space. By doing this the popping is removed, or at least less noticeable. However, the downside of this technique is that rendering two meshes for a single object at the same time can be a waste of hardware resources. A more detailed description can be found in section 3.2.

Both the mentioned techniques only work with a set of static models. There are other types of techniques that combine this with different methods of generating additional geometry of those models, effectively creating a much higher number of models to use. This method is called adaptive or dynamic LoD and is applied during run-time based on different criteria, like distance from the screen or screen-area occupied.

One such technique is called Curved PN Triangles [4], which generates more detail uniformly over the whole mesh. Another technique is called Phong Tessellation [5]. This technique is focusing on generating more detail on the silhouette only, which is considered one of the most important features of an object [11, 12, 13]. The details of these techniques can be found in section 4.1.

The four techniques mentioned above are some of the most commonly used for games, and are implemented in several of the most popular game engines used today. Static LoD is used in most games because of its simplicity and low performance requirements. Unpopping LoD was not specifically mentioned by name in the sources found, but many use LoD blending for smooth transitions, which is what Unpopping

LoD does. Curved PN Triangles is implemented in both CryEngine [32] and Unreal Engine [34]. Both CryEngine [32] and Unity [35], as well as the game Metro 2033 [33], implements Phong Tessellation.

There are many other techniques that can be used to generate more geometry. There is, for example, Butterfly scheme [6], Catmull-Clark subdivision [7], $\sqrt{3}$-subdivision [8], Loop subdivision [9], and many more. There are also a number of variations and additions to these techniques, like Scalar Tagged PN Triangles [10]. Catmull-Clark and Loop subdivision is commonly used in the movie industry through the library OpenSubdiv created by Pixar [36]. No sources were found where these two, or the other techniques mentioned above, were used for games. Therefore they were excluded from this study.

The papers [3, 4, 5] all show comparisons of performance using measurements of, for example, Frames per Second (FPS), number of vertices/triangles, and memory usage. There are very few who use image space metrics to measure the difference, both perceptual and pure pixel data, of the resulting images of the different techniques. This means that it is difficult to determine if a technique is suitable to use. The performance of technique A might be better than that of technique B, but if B has much better visual result then A might not be worth using. With a comparison of the image space result it would be easier for developers to choose the technique that best suits them. This project will try to bridge this gap and provide both performance comparisons and comparisons of image space metrics.

1.2 Aim and Objectives

The silhouette and contours of the object is used in object recognition and object-to-ground separation in the human brain, which makes it one of the most important features [11, 12, 13]. With this in mind, Phong Tessellation should be a suitable solution to use for Level of Detail, as it would use the resources on the areas of highest importance. The uniform refinement of Curved PN Triangles is expected to use more resources refining areas of low importance.

The aim of this project is to determine how the silhouette-based Phong Tessellation performs compared to Curved PN Triangles, Unpopping LoD, and traditional Static LoD in the context of both performance and image space metrics.

The objectives are:

1. Create a real-time application for handling of objects, shaders, rendering, and logic such as camera movement.
2. Implement the techniques Static LoD, Unpopping LoD, Curved PN Triangles, and Phong tessellation.
3. Gather the performance measurements and the images that will be used for the image space comparison.
4. Perform the experiments, evaluate the results, and draw conclusions

1.3 Research Questions

Considering the LoD techniques Phong Tessellation, Curved PN Triangles, Static LoD and Unpopping LoD, and the image space metric developed by H. Yee [14],

RQ1: How do the differences in image space compare considering the continuous quality of the images and the switching between meshes?
RQ2: How do the performance metrics (FPS and triangle count) compare when rendering using these LoD schemes?

2 TESSELLATION AND THE GRAPHICS PIPELINE

The tessellation functionality is the newest addition to the modern graphics pipeline, which was added in Direct3D 11. This new version supports three new stages that are used to implement tessellation, which is used to convert low-detail primitives, such as triangles, into higher detailed primitives on the Graphics Processing Unit (GPU). By implementing tessellation in hardware the graphics pipeline can evaluate and improve the quality of a low-detail model, and render it in higher detail. An example of this can be seen in Figure 2 below. A low detailed mesh is sent to the renderer, tessellation is performed, and the final result is a cube with many more triangles.

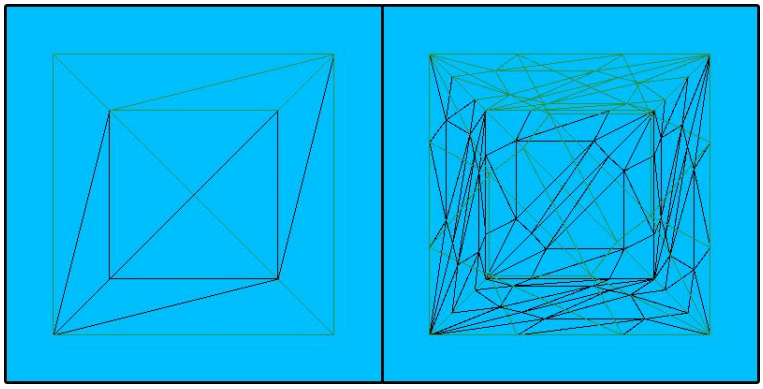

Figure 2. An example of how a low-detail mesh (to the left) is subjected to tessellation (to the right).

To understand how the tessellation itself works, this chapter will provide a quick rundown of the graphics pipeline and its stages, with a deeper focus on the new tessellation stages.

When rendering an object, the data that is used to represent the model is sent through the graphics pipeline where various effects like shading and texturing is applied, and ultimately it is transformed into the two-dimensional space of the screen.

The pipeline consists of several programmable stages, each stage with its own domain of tasks to perform. These stages can be seen in Figure 3 below. The Geometry Shader Stage and the Stream Output Stage were omitted from this chapter because they are not used in this study and therefore are not relevant to this topic.

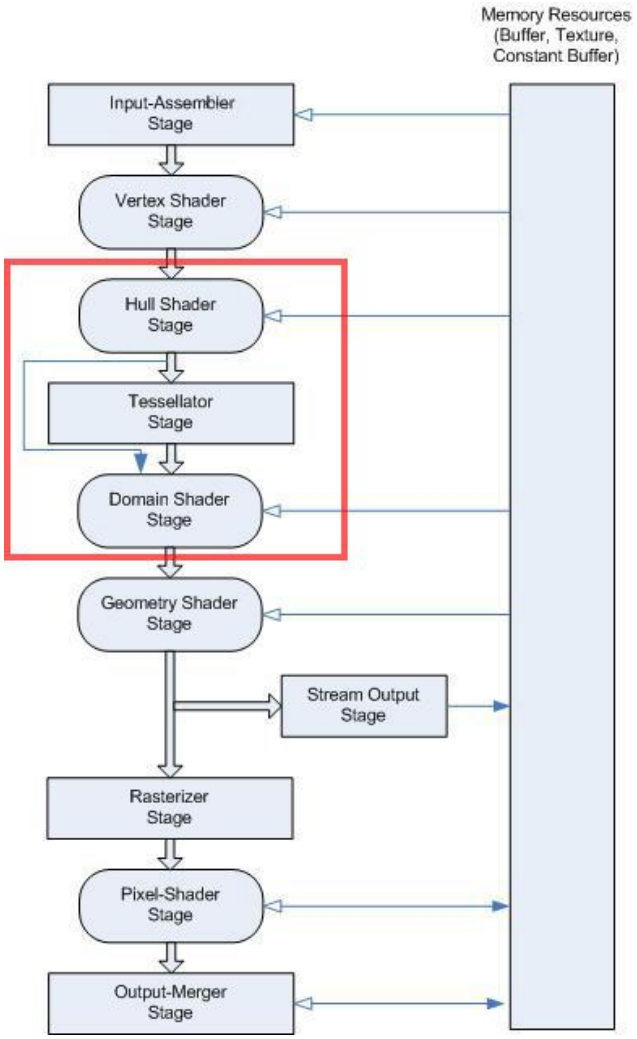

Figure 3. The graphics pipeline and its stages. The tessellation stages are highlighted with the red square. [30]

2.1 Input-Assembler stage

The entry point of the pipeline is the input assembler stage. This stage is responsible for reading the input data of the mesh. A mesh is made up of vertices, which are points in the three-dimensional model space. These vertices are assembled based on the chosen topology, such as a list of triangles, for use later in the pipeline.

2.2 Vertex Shader Stage

This stage reads the assembled vertex data from the input assembler stage and processes one single vertex at a time. Every vertex is processed, and is done so without the information of the neighboring vertices. One of the usual responsibilities of this stage is to transform the vertices to clip-space, which is the two-dimensional space of the screen. However, when tessellation is applied it is done in a later stage.

This stage can only process already existing vertices, which is why tessellation is an important addition to the shader pipeline.

2.3 Hull Shader Stage

This is the first of the three stages used to implement tessellation. The hull shader receives data from the vertex shader stage in the form of *patches*, consisting of *control points.* A patch can, for example, be a triangle, where the control points are the three corner vertices.

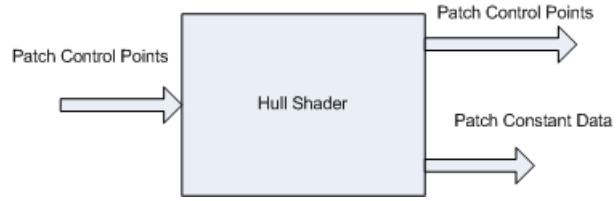

Figure 4. A simple example visualizing the input and output flow of a hull shader. [31]

The hull shader has two major responsibilities. The first is to determine how the patch should be tessellated. This is done using a function in the hull shader which is run once per patch. At the very least this function must provide a set of tessellation factors, but it can also include other user-defined data necessary for the later parts of the pipeline. The hull shader also selects different settings that determines things like which tessellation algorithm should be used, or which topology the output of the tessellation should have.

The other responsibility is that it has to create the control points that will later be used by the domain shader stage. This is done once per control point in the hull shader.

The input and output of this stage is visualized in Figure 4 above. The output Patch Control Points are the control points that it creates for use in the domain shader, and the Patch Constant Data is the tessellation factors and other user-defined data.

2.4 Tessellator Stage

The tessellator stage is not programmable like the other stages, but instead it uses one of several algorithms to perform the tessellation. Which of these algorithms it should use is specified by the hull shader. Depending on the tessellation factors and the chosen settings of the hull shader it will determine which points in the current primitive it needs to use to divide it into smaller parts.

The output of this stage is in barycentric coordinates. Barycentric coordinates (u,v,w) can be seen as the area of sub-triangles CAP for u, ABP for v, and BCP for w over the area of the triangle ABC. An example of this for a point P can be seen in Figure 5 below.

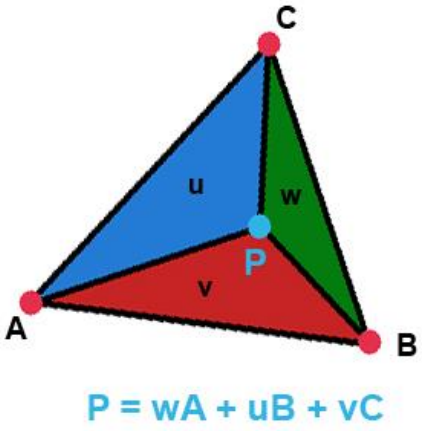

Figure 5. An example of how a point P is expressed in the barycentric coordinate system

2.5 Domain Shader Stage

This is the last of the tessellation stages. The domain shader receives the barycentric coordinates from the tessellator along with the control points and user-defined data produced by the hull shader, and uses it to create new vertices. Often there is some functionality implemented to offset the new vertices in some way to create a slightly different shape, like smoothing out hard edges and corners.

This stage is a suitable place to apply the transformation to clip-space which was mentioned in section 2.2.

2.6 Rasterizer Stage

After the data has passed through the Domain shader (or the Geometry shader if used) the portion of the pipeline that operates on geometric level is completed. In this stage the data is rasterized and handled at fragment level. A *fragment* is a group of data that corresponds to a pixel which has the possibility of being shown on the screen if it passes through the next stages. Each fragment receives interpolated pixel values based on the vertex data. In addition to this it also generates a depth value that will be used in later stages to determine the visibility of each fragment.

The rasterizer is, like the tessellator, a fixed-function stage. This means that is not directly programmable, but has different states and settings that can be set by the programmer beforehand.

2.7 Pixel Shader Stage

When the fragments are generated from the rasterizer they are sent to the pixel shader. This stage is responsible for per-pixel operations, such as the coloring of the pixels. It can be as simple as applying a color value, or more advanced methods like sampling and applying one or more textures. This is also the stage where lighting and shading is most often applied.

2.8 Output Merger Stage

This is the final stage of the pipeline, and here the output merger is responsible of merging the data to the bound resources like render targets and depth stencils, performing the blending functions, and performing the actual writing to the resources. When the data is written to the resources the image is done and ready to be displayed on the screen.

3 THE TECHNIQUES

The four techniques used in this study were briefly mentioned in the introduction, but to understand more about how they work, this chapter will explain them further.

3.1 Static LoD

Static LoD is the simplest technique to use when working with Level of Detail, and is based on the proposal made by J. Clark [1]. The mesh to render is decided based on some criteria, like distance from the in-game camera. For example, a simple setup might look like in Table 1 below. LOD0, LOD1, and LOD2 are the different meshes used for the object, where LOD0 is the most detailed one.

	LOD0	LOD1	LOD2
Distance interval	0 – 10	11 – 20	21 – 30

Table 1. An example of how the intervals for LoD-selection is set up.

If the distance is 10 it means that LOD0 will be used. The in-game camera is then moved further away from the object. When the distance goes from 10 to 11, mesh LOD1 will be used instead. This hard switch introduces the problem mentioned earlier called popping. Figure 6 below shows pictures from the game Tom Clancy's The Division™ where popping is visible.

Figure 6. Images of in-game footage of Tom Clancy's The Division™, developed by Massive Entertainment. Released in 2016.

3.2 Unpopping LoD

Unpopping LoD [3] is utilizing the blending functionality of the hardware, which is applied in the output merger stage, to try to hide the popping that occurs when switching between meshes. It does so by rendering both the current mesh and the one that will be switched to at the same time during an interval of, for example, time or distance.

Before this technique was developed, there was another commonly used technique that tried to solve the popping through blending. However, the problem with the way that technique handled the switch between meshes was that during the interval both of the objects were rendered semi-transparently. If an interval of distance was used, and the player stopped moving somewhere during that interval, the objects would stay semi-transparent. If the interval was instead time-based there was no sweet-spot for the blend-time. Either the blend-time was short enough so that the user would not notice the semi-transparency of the object, but instead there would be visible popping, or the blend-time was longer which would remove the popping but instead introduce the problem with semi-transparent objects. The problem with transparency is visualized in the left part of Figure 7 below.

Figure 7. The left part shows the "incorrect" way of blending, where the objects are semi-transparent. The right part shows how it should be done. [3]

The developers of Unpopping LoD found a way to handle the blending which does not result in transparent objects, but instead has the result visualized in the right part of Figure 7 above. What they do is that during the interval, one of the two meshes will be opaque and write to the depth buffer at a time. If switching between mesh LOD1 to LOD2, then LOD1 would be opaque and perform depth buffer writes during the first half of the interval. During the second half LOD2 would instead be the mesh used for the opaque rendering and depth writes. This setup is shown in Table 2 below. Doing it this way ensures that the background will never shine through the object, while at the same time removing, or at least reducing, the popping.

	Unpopping LoD				Old technique	
Interval	$t \in [0, 0.5]$		$t \in [0.5, 1]$		$t \in [0, 1]$	
	LOD1	LOD2	LOD1	LOD2	LOD1	LOD2
Alpha value	1	2t	2(1-t)	1	t	1-t
Depth test	true	true	true	true	true	true
Depth write	true	false	false	true	false	false

Table 2. The setup used during the interval of the Unpopping LoD technique. The old technique is included as a comparison.

3.3 Curved PN Triangles

The aim of this technique [4] is to improve the visual quality of the objects in games by smoothing out the shape of an object, creating a less edgy silhouette and providing more points to sample from when performing vertex operations. To do this the authors introduce *curved point-normal triangles*, or *PN triangles* for short. Each PN triangle replaces a simple flat triangle by a curved shape that consists of many smaller sub-triangles.

The geometry of each PN triangle is defined as one cubic Bézier patch [28], which matches the vertex and normal data of the original flat triangle. The PN triangle consists of several control points that are used to create the new curved surface. These control points are interpolated values of the vertex data of the flat triangle, and are used to create new vertices. An example of a PN triangle can be seen in (a) in Figure 8 below.

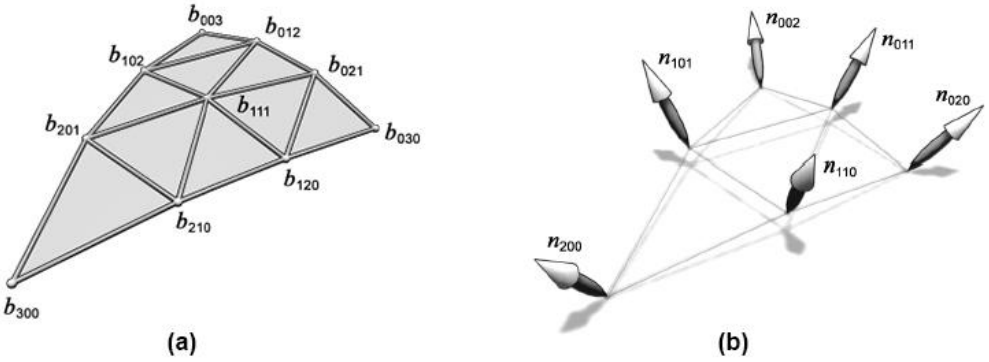

Figure 8. (a) Generated control points of a PN triangle. (b) The control points of the normal component of a PN triangle [4]

For the new surface to be correctly displayed and shaded, it also needs to take the normals of the flat input triangle and interpolate them over the new sub-triangles. This can be done either as a linear interpolation of the vertex normals, or as a quadratic function of the position and normal data. (b) in Figure 8 above shows the control points used for the normal interpolation.

When the interpolations of both positions and normals are done, the final object will have a smoother surface with more detail and smoother silhouette. Figure 9 below show the result (c) after the Curved PN Triangles-technique is applied to an input mesh of low detail (a, b).

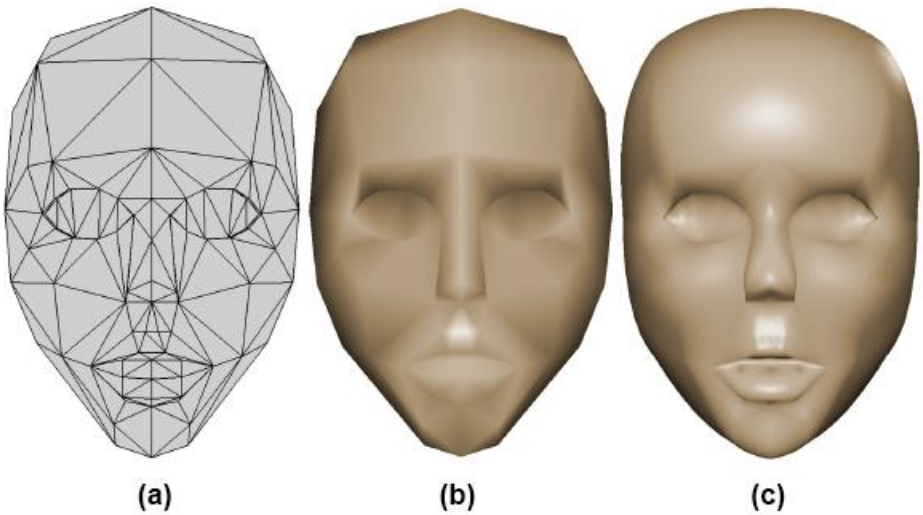

Figure 9. (a) Input mesh of low detail in wireframe mode. (b) Input mesh of low detail in goraud-shaded mode. (c) Curved PN Triangles applied to the input mesh, resulting in a more detailed result.[4]

3.4 Phong Tessellation

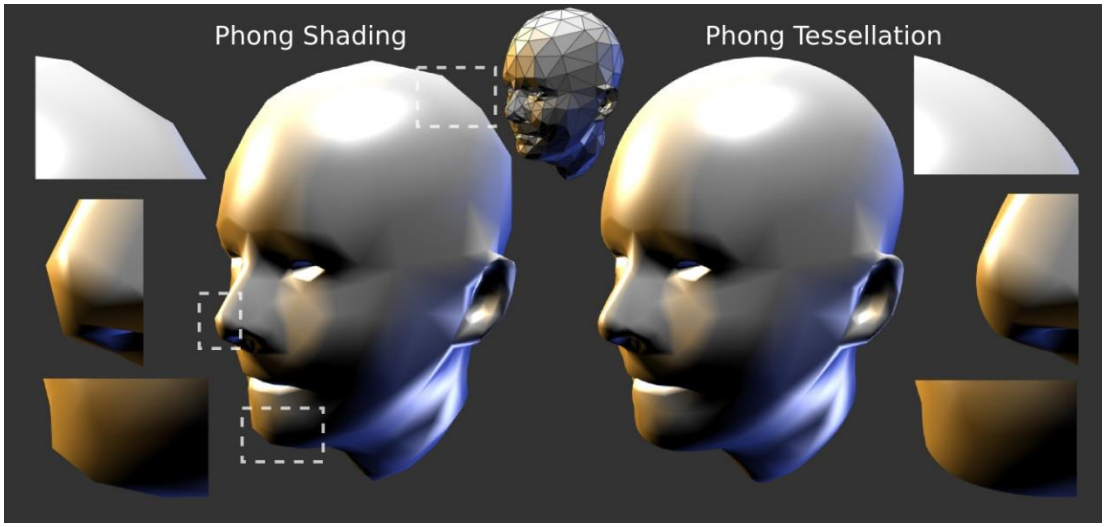

Figure 10. Phong tessellation removes the artifacts by smoothing the silhouette. [5]

Phong Tessellation [5] is based on the same concept as Phong normal interpolation, which is used to interpolate normals to provide better shading. Even if better shading improves the quality of the visual result, if the geometry is not smooth artifacts will remain at the silhouettes and contours. Phong Tessellation tries to solve this by performing a geometric interpolation of the vertex positions, which provides more geometry and detail to remove the artifacts on the silhouette and contours. The result can be seen in Figure 10 above. The left part shows the object with only Phong shading applied, and highlights the artifacts caused by the course silhouette. The right part shows the same Phong shaded object with Phong Tessellation applied, and highlights the now smooth silhouette.

According to the authors there are two competing goals when trying to improve the quality along the contours:

- Producing smooth geometry along contours so that the visual artifacts can be avoided
- Creating this geometry with as few operations as possible, as the area around the contour only accounts for a small part of the image.

Phong tessellation was designed with these two points in mind, making it less expensive than many other techniques used to refine the silhouettes and contours.

It is based on barycentric interpolation and orthogonal projections, which is illustrated in Figure 11 below. This requires only the information already carried by a triangle, namely the vertex positions and the vertex normals.

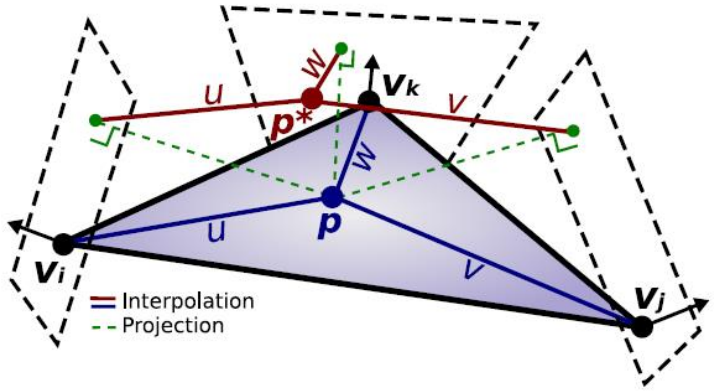

Figure 11. Interpolates the projections onto vertices tangent plane to provide a curve geometry for each triangle. [5]

4 METHODOLOGY

To perform this study an application with suitable rendering functionality was implemented. This application also has the functionality to gather the performance measurements and the images used for the experiment. The experiment consists of two parts. One part concerning the performance and one part concerning the image space comparisons. More details about the experiment are found later in this chapter.

Information and references were found by conducting searches in Google Scholar and databases like ACM Digital Library and IEEE with different search strings, like "level of detail for games". Some tips about papers were provided by the supervisor.

4.1 Implementation

To be able to implement and test these techniques, an application with the functionality for the handling of objects, rendering, shaders, and some logic such as moving the camera and rotating the object was created. This was done using Microsoft Visual Studio 2015 Community Edition, C++, and Direct3D version 11 together with Shader model 5.0 to support the tessellation functionality. The output target of the rendering has a resolution of 900x900 pixels.

All four techniques were implemented in the same project, with an easy way of switching between them. This is to ensure that all techniques has a common base, without differences other than those created by the techniques.

The scene consists of a single object and an in-game camera, which basically is the view from which the scene is rendered. As Phong tessellation is view-dependent, the object needs to be seen from all directions and distances. This is accomplished by having the camera moving towards the object while rotating around it using spherical coordinates [29], lowering the radius to move closer to the object. Moving closer to the object is necessary to trigger the LoD-switches, which are distance-based. A visualization of the camera movement can be seen in Figure 12 below.

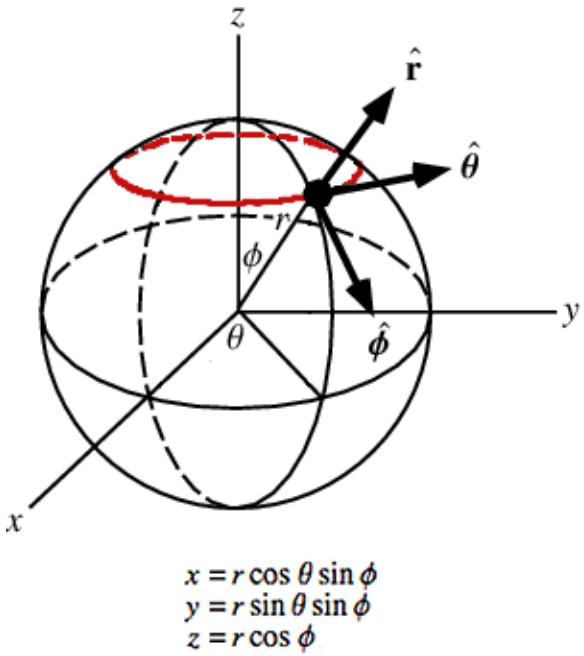

Figure 12. Visualization of the camera movement around the object. Modified version of the original image found in [29].

The black dot is the camera position, which is given by the calculations seen in the lower part of the image. Modifying the angles θ and ϕ will result in different positions on the sphere. For example, if θ is increased, the point will move counter clockwise, following the path highlighted by the red circle. If ϕ is increased the circle will move further down the sphere.

Modifying the radius, represented by r in the image, will make the sphere bigger, and therefore increase the distance between the camera and the object.

The object is positioned in the middle of the sphere, and the camera turns toward it when moved around the circular path.

4.2 Experiments Setup

The experiments will be conducted using data gathered by the application. The number of meshes that will be used for a single object is five, and the average decrease of triangles between them is 50%. Both these numbers are based on research of the gaming industry, and which numbers they use. For example, in the official documentation for the widely used game engine CryEngine [15], they explain that developers should reduce the number of triangles of each level by 50%. They also inform that no more than six levels of meshes are supported. Other sources from the industry [16, 17, 18, 19] have similar numbers. The number of meshes used is in the range of three to eight, and they uniformly agree that 50% is a good percentage of reduction to aim at, but at least somewhere between 30%-66% should be used.

The geometric complexity of objects in games vary greatly. According to [20, 21, 24, 23, 24], the number of polygons of objects such as the main character can be anywhere in the range 5,000-100,000+. However, the most common amount to use is around 40,000-50,000. The objects used in this study will therefore be in that range. More information about the objects used can be found in chapter 5.

The ranges for the different levels are adapted to the pattern used for the reduction percentage, as this is a good match with the geometric complexity. A 50% decrease of complexity gives 2x the range of the previous level. However, this is a completely different area of research, and there is no "correct" way of doing it. Even so, the techniques all use the same ranges, which means that the measurements will depend on the techniques, not the range used to decide mesh level. The ranges (measured in arbitrary units) can be seen in Table 3 below.

LOD0	LOD1	LOD2	LOD3	LOD4
0-8	8-16	16-32	32-64	64+

Table 3. LoD-ranges used in the experiments.

4.2.1 Performance

When comparing different techniques there are many factors affecting the performance. A common measure in games and real-time applications is Frames per Second (FPS), as this measure provides a number which represents the general performance of the whole application. The FPS is also most relevant to the user, as this is what will be noticed, and it is easy to see if it is within the acceptable range.

The comparisons in this project will use FPS along with the number of rendered triangles as factors. Knowing the number of triangles is interesting since it will show the results of, for example, the view-dependence of Phong tessellation and the double rendering during the switch of meshes using Unpopping LoD.

These factors will be gathered by the application and summarized to show the differences, and similarities, of the techniques.

4.2.2 Image Space

The images used for these experiments will be saved by the application, copying the contents of the render target, and will then be compared by the Perceptual Visual Difference Utility developed by H. Yee, using the metric he proposed [14]. His metric is a model that tries to emulate the human visual system as close as possible, and provides a way to compare the perceptual difference between two images. This difference is referred to as the error between the images. A lower error means lower perceptual difference, and a higher error means higher perceptual difference.

The error will be measured in two parts:

- The error caused by the switching of meshes, in other words the popping.
- The error of consecutive images compared to the same image rendered with the static mesh of higher quality and the static mesh of highest quality.

The first part is measured by comparing the image just before the switch and the image just after the switch. An example of this can be seen in Figure 13 below. For Unpopping LoD it is done by taking the whole interval of blending into account, calculating the minimum, maximum, and average errors.

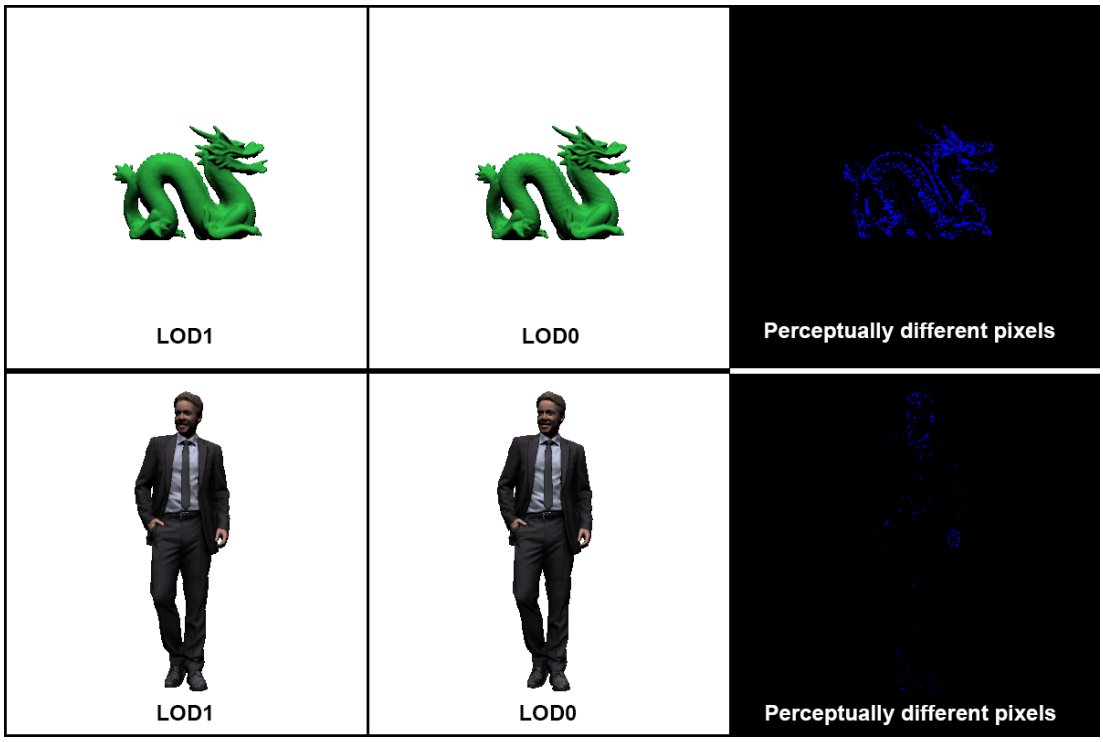

Figure 13. The error between the frame just before the switch (left), and the image just after the switch (middle) is visualized by the blue pixels in the right image. This example is using Static LoD.

The second part is measured by calculating the error of every image compared with an image rendered from the same perspective using the static base mesh of the level above in quality. The goal for every level is to resemble the level above as close as possible, which makes this the ideal case. For example, the result for a dynamic technique where the improved mesh L01 (using L0 as base) is used is compared to the same image when L1 (base mesh for the level above) is used without any geometric improvement. A visualization of this can be seen in Figure 14 below. A gradual improvement in quality should result in less popping caused by the switching of base meshes.

To determine if the dynamic techniques generate more detail than the next level, comparisons with the highest level will also be included. If the hardware was not limited, the highest level would be used at all times, which technically means that this is the ultimate goal of every level.

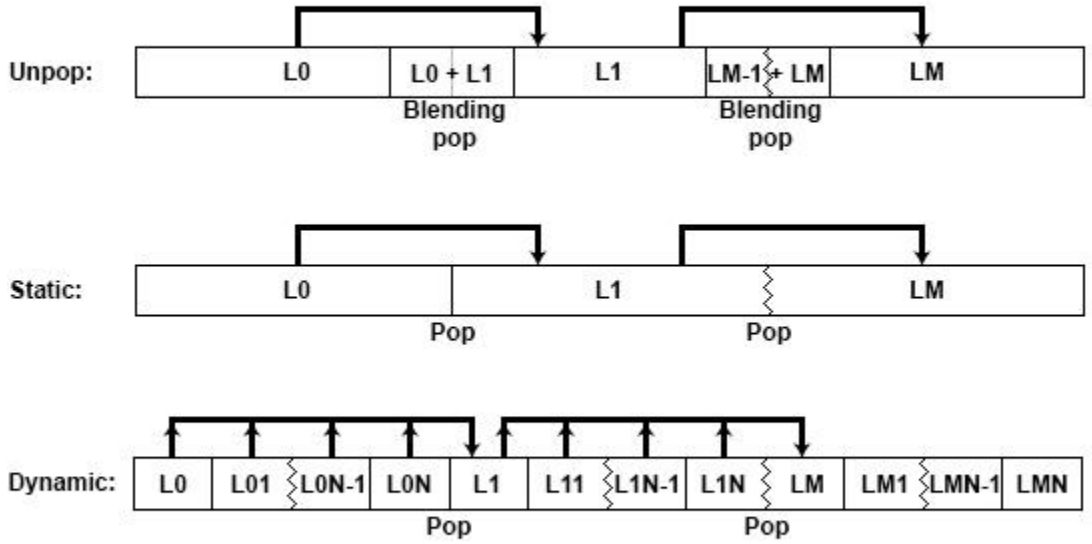

Figure 14. Shows how the images will be compared. For example, the improved mesh L01 (using L0 as base) is compared to the same image when L1 (base mesh for the level above) is used without any geometric improvement.

4.3 Hardware

The experiments will be conducted on a computer with the following hardware:

CPU	Intel® Core™ i7-4700HQ CPU @ 2.40GHz (8 CPUs), ~2.4GHz
GPU	NVIDIA GeForce GTX 780M. Driver version: 365.10
RAM	16GB DDR3
OS	Windows 10 Home 64-bit, build 10586

Table 4. The hardware specifications of the computer where the experiments will be conducted.

5 THE EXPERIMENTS

This chapter will present the results of both of the experiments, which were conducted according the description in section 4.2. The experiments were performed using three different models with different characteristics to determine how the dynamic techniques handle different geometric shapes. The first one was a textured model of a man in a suit and the second one was the Stanford dragon. Both of them have shapes with smooth curves and natural formations. The third one was a semi-truck with a trailer, which consists of many unnatural hard shapes and edges. The three models can be seen in Figure 15 below.

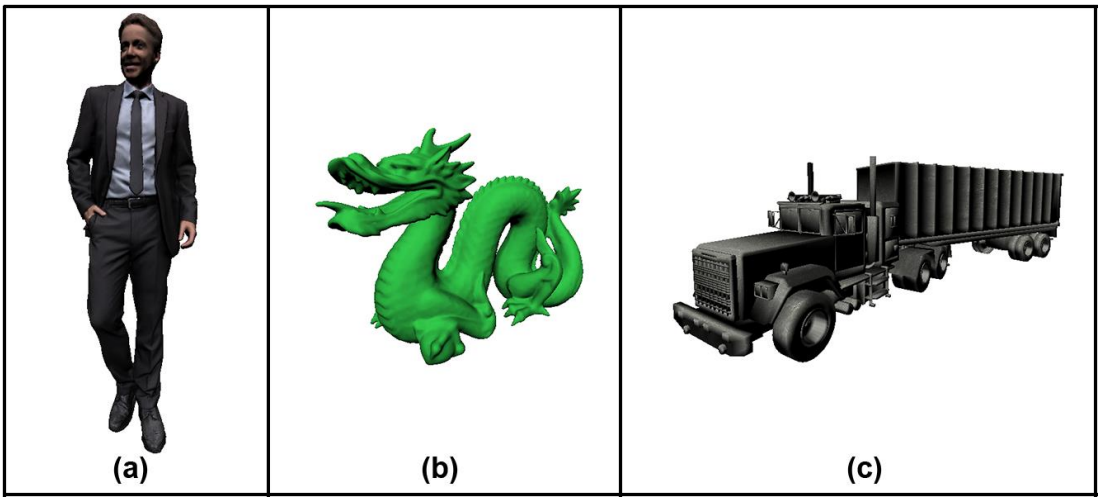

Figure 15. The models used for the experiments. (a) Textured man in a suit. (b) Stanford dragon. (c) Semi-truck with a trailer

The original model of the Textured Man [27] has ~100,000 polygons, but was modified to conform to the guidelines of geometric complexity used. LOD0 has 50,000 triangles and the subsequent levels were generated with a reduction of 50%.

The Stanford dragon model was downloaded from the Stanford University Scanning Repository [25], and is often used by the industry to showcase different techniques or technology. The original has 871,414 triangles, but LOD0, which is the level with the highest level of detail, has only 49,386 triangles in these experiments to conform to the guidelines regarding geometric complexity. The subsequent levels were generated with 55% reduction.

The original model of the semi-truck with a trailer [26] has ~77,400 polygons, but was modified some to remove parts with a high density of geometry. LOD0 has 44,872 triangles. The subsequent levels were generated with a reduction of 50%.

5.1 Results

When the experiment was performed with the semi-truck with a trailer it quickly showed that the dynamic techniques used in this study do not work well with hard shapes and edges, smoothing them and deforming the object until some parts were barely recognizable. This can be seen in Figure 16 below by looking at the exhaust pipes. This was a consistent behavior on all levels of detail of the object. Because of this, no further experiments were performed with this object.

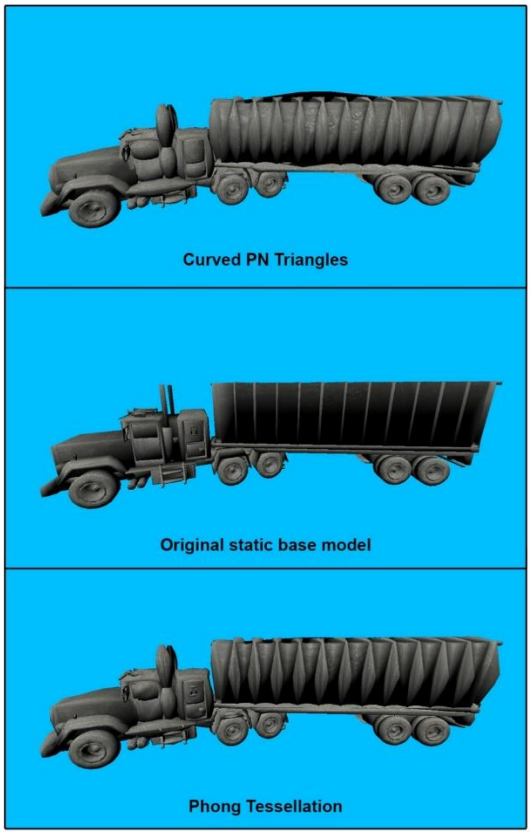

Figure 16. Shows the deformation and artifacts created when applying the dynamic techniques on an unnatural object with many hard edges and shapes. The base mesh used in this image is the second most detailed level with ~22k triangles.

The other two objects did not show such deformations and artifacts, and the experiments were performed as planned. All measurements and comparisons of the gathered data are presented in easily viewable graphs. They are also summarized in Table 5 and Table 6 in section 5.1.4.

5.1.1 Performance experiments

Since the semi-truck with a trailer was unsuitable for use with the techniques using tessellation, the performance was only measured with the dragon and the textured man. The experiment was conducted as explained in section 4.2, where the camera moved closer to the object while rotating around it.

Capture points are the points in time that the data was gathered. The number of frames was captured once every second. Because the number of triangles and the FPS is closely connected, the number of triangles should be captured at the same time. However, capturing the number of triangles only once per second did not show as clear result, since much can change during that second. Because of this, the number of triangles was captured every half second, which is used for the X-axis named “Capture points” in the graphs. Every “Capture point” is a half second.

The differences between the models themselves can be explained by looking at the number of triangles, which for the Man is higher than the Dragon. The Man is also textured, which means that some extra time is spent on sampling and applying the texture.

5.1.1.1 Static LoD

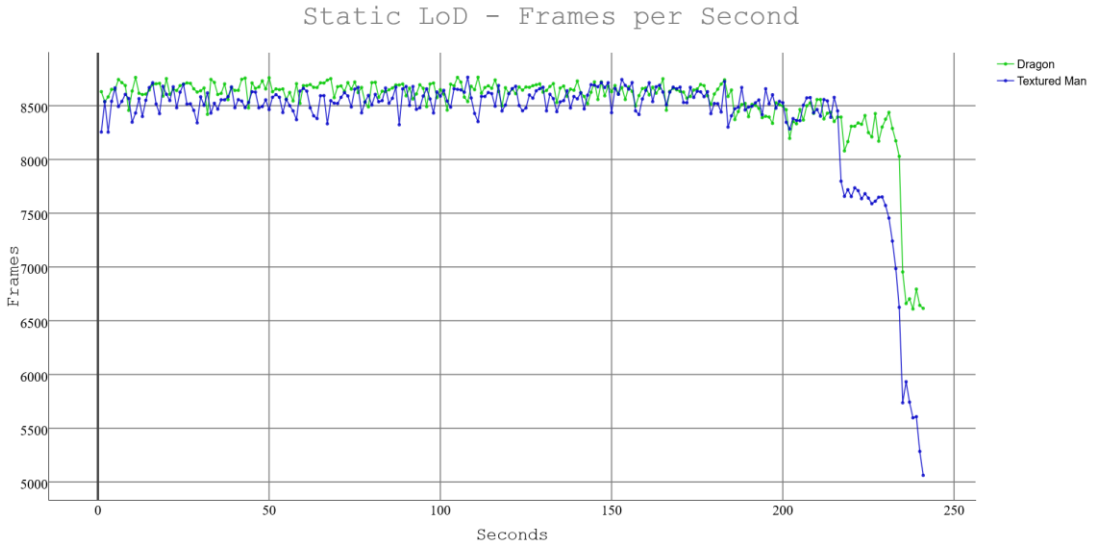

Figure 17. Frames per Second gathered when rendering with Static LoD

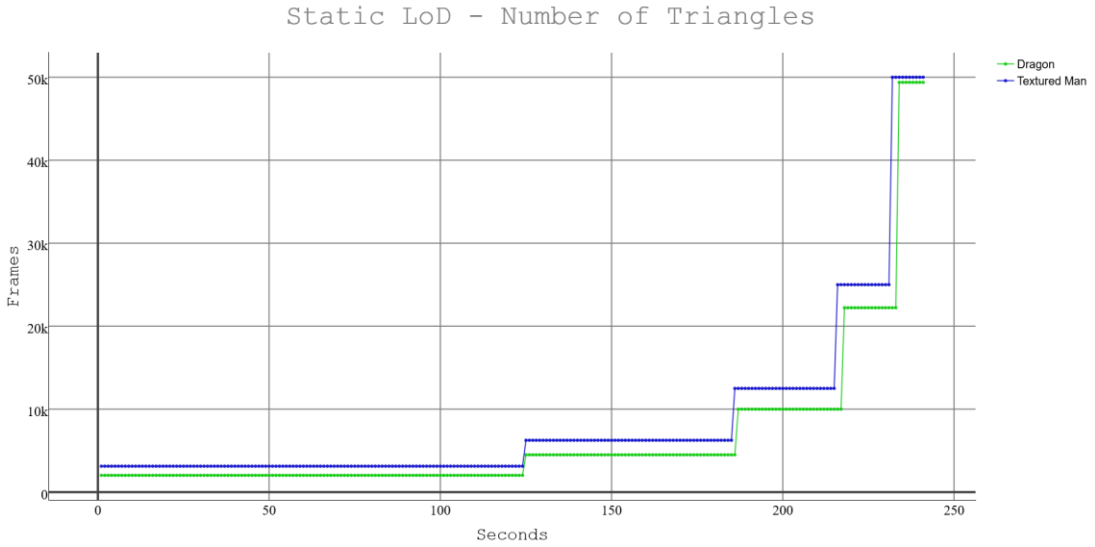

Figure 18. Number of rendered triangles gathered when rendering with Static LoD

The FPS follows a similar pattern for both models using this technique. The Man has slightly lower FPS, otherwise there are no significant differences.

The chart with the number of triangles shows no unexpected results, since Static LoD works with the static base meshes only. However, it is included to be able to compare to the other techniques.

5.1.1.2 Unpopping LoD

Figure 19. Frames per Second gathered when rendering with Unpopping LoD

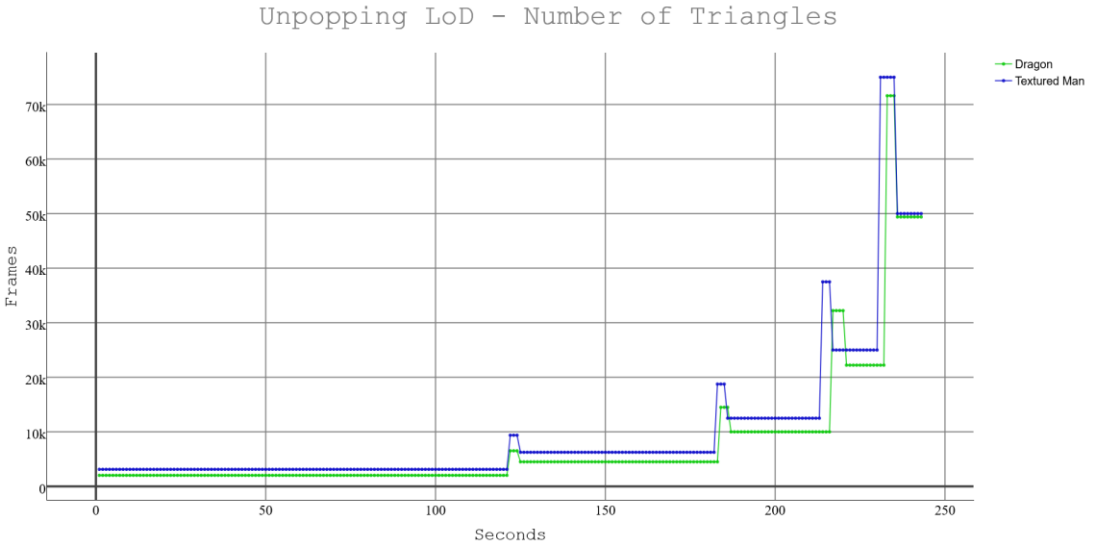

Figure 20. Number of rendered triangles gathered when rendered with Unpopping LoD

Unpopping LoD shows similar results compared to Static LoD, but during the blending interval of the switches there are some differences. Looking at the number of triangles it is clear that the technique renders two meshes at the same time. During this double rendering we can also see that the FPS drops significantly. For example, when switching between the last two levels of the Textured Man, the FPS goes from ~7,500 before the interval starts to ~4,600 during the interval, and then up to ~6,700 after the switch. During that time the number of triangles goes from 25,000 up to 75,000 during the interval, and then down to 50,000 after the switch.

5.1.1.3 Curved PN Triangles

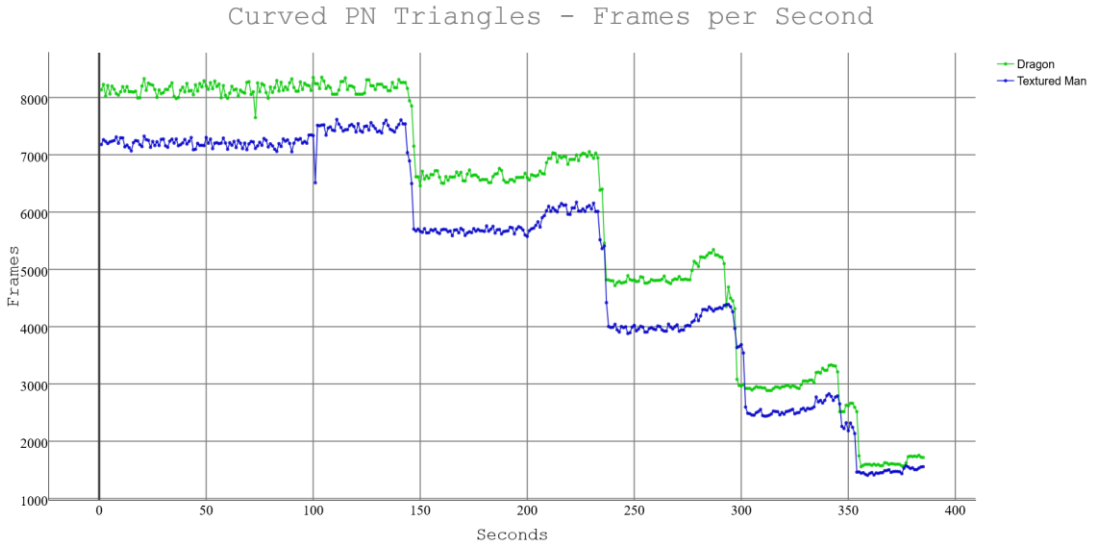

Figure 21. Frames per Second when rendering with Curved PN Triangles

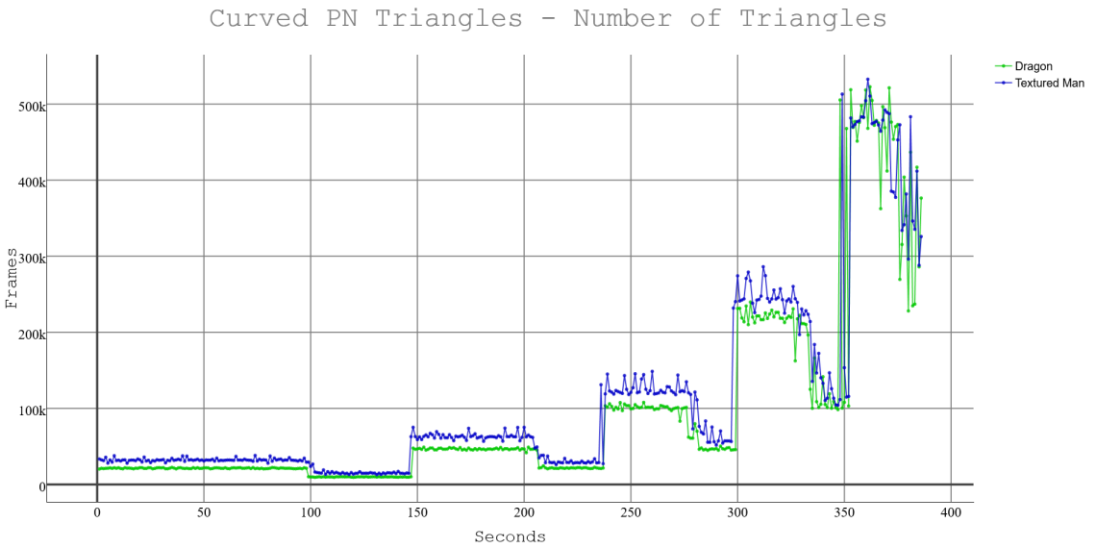

Figure 22. Number of rendered triangles gathered when rendering with Curved PN Triangles

The FPS for both models shows a similar pattern, where it drops significantly for every switch. Looking at the number of triangles we can see that this drop corresponds to the increase of triangles. This increase is much bigger than both Static LoD and Unpopping LoD because of the tessellation. As the base mesh is increased in number of triangles, the tessellation will also increase according to that, which is why there is a clear pattern for the number of triangles as well. This also explains why the FPS is lower using this technique, compared to Static LoD and Unpopping LoD.

The drop in number of triangles before the switch can be explained by the angle the object is seen from. During this time it is seen from above, which means that for both objects there will be fewer triangles facing the camera. The triangles not facing the camera will be removed before the tessellation, and therefore fewer triangles will be tessellated.

5.1.1.4 Phong Tessellation

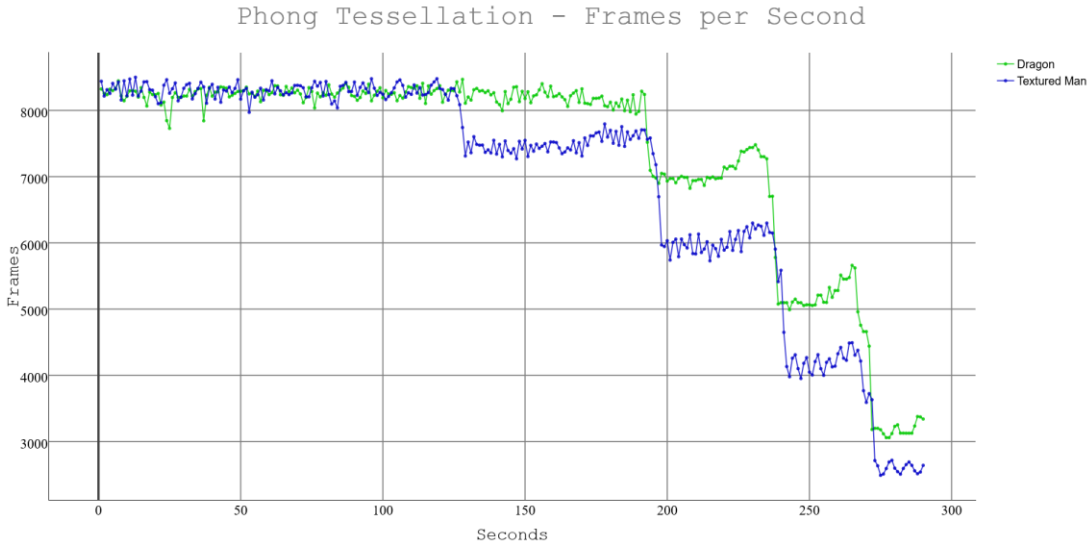

Figure 23. Frames per Second when rendering with Phong Tessellation

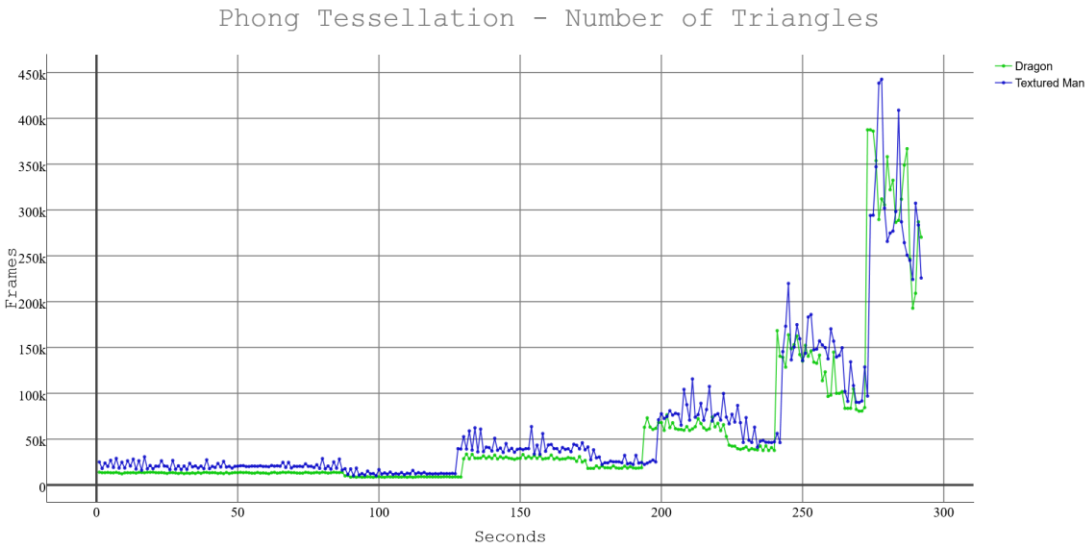

Figure 24. Number of rendered triangles gathered when rendering with Phong Tessellation

The results from this technique has similar patterns compared to Curved PN Triangles. However, the silhouette-focus clearly shows a difference in the number of triangles, which also affects the FPS. Less triangles are created because the focus is on the silhouette, and therefore the FPS is higher.

The drop in number of triangles before the switch happens for the same reason as for Curved PN Triangles, it depends on the viewing angle. This is also the reason for the spikes in number of triangles. During these spikes the object was seen from angles with much silhouette visible, which means more refinement is applied.

5.1.2 Continuous error

During these experiments the camera moved towards the object while rotating around it, as in the performance experiments. The higher errors in the end can be explained because of the size of the object on the screen. When it covers more area of the screen there are more pixels that can potentially be an error. In the beginning the object occupies a very small part of the screen, and therefore the error will be much smaller.

The camera was moved some, and then the image was captured for all four techniques. The “Images” in the X-axis of these graphs represent these gathered images. For example, image 1000 will have exactly the same orientation of the camera and the object for all four techniques.

All of the techniques show a similar pattern in which the error increases as the camera moves closer, and drops somewhat when the mesh is switched, then starts increasing again. This pattern is visible in both the comparison with the level above (blue lines), and with the highest level (green lines). The errors are also varying a lot between consecutive images, where it can be, for example, 2,500 in one, and 1,000 in the next. This is because of the rotation, and the fact that the error depends on how much of the object that is visible. For example, when looking at the man from above it will have fewer errors than if looking at the man from the front.

The Dragon also shows errors of much higher numbers, which can be explained by looking at Figure 13. This figure shows where the error is found, and for the Man most of the error can be found on the silhouette. The Dragon has much more shapes within the object with bumps and scales on its body, and when the level of detail changes on them it will result in errors all over the mesh. The error on the silhouette is added to this as well.

Towards the end, the green lines from the comparison of the highest level seem to disappear. This is because the level above and the highest level is represented by the same mesh, which leads to the exact same error.

5.1.2.1 Static LoD

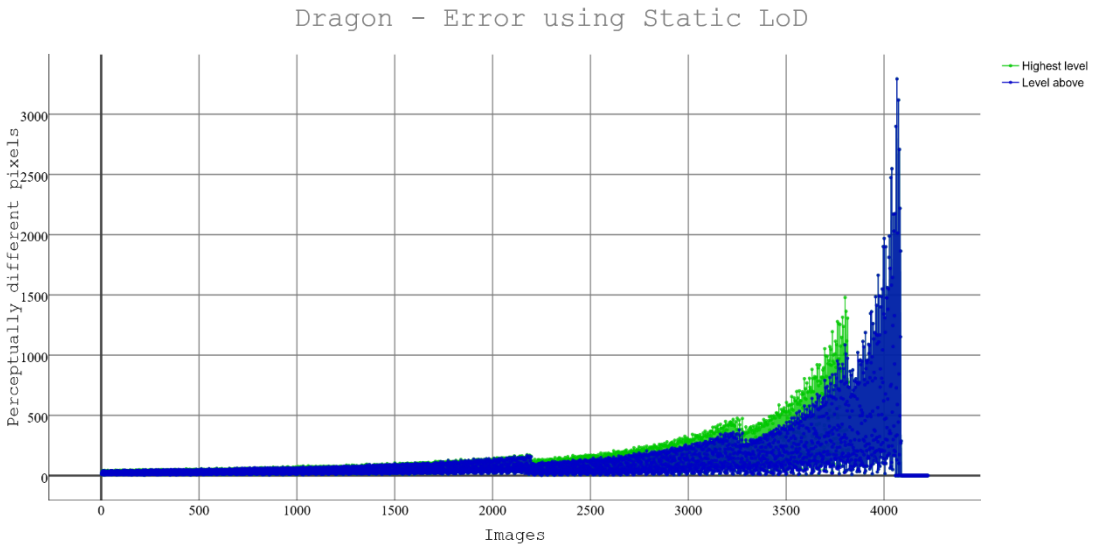

Figure 25. Continuous error of the Dragon using Static LoD

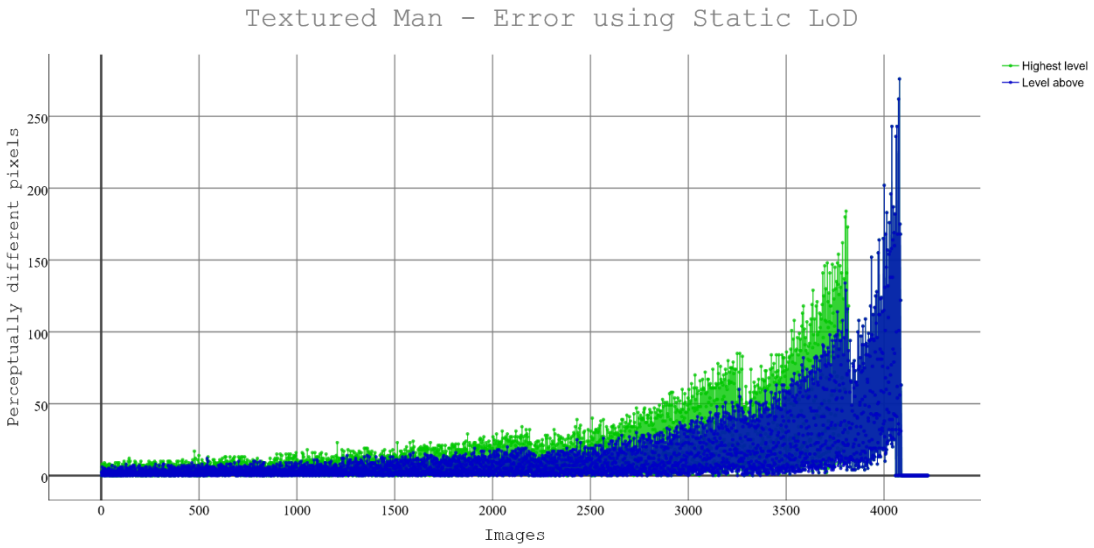

Figure 26. Continuous error of the Textured Man using Static LoD

The error for both models, compared to both the level above and the highest level shows the same pattern, as described in the previous section. In the end there is no level above and LOD0 is compared to LOD0, which explains the error dropping to 0.

5.1.2.2 Unpopping LoD

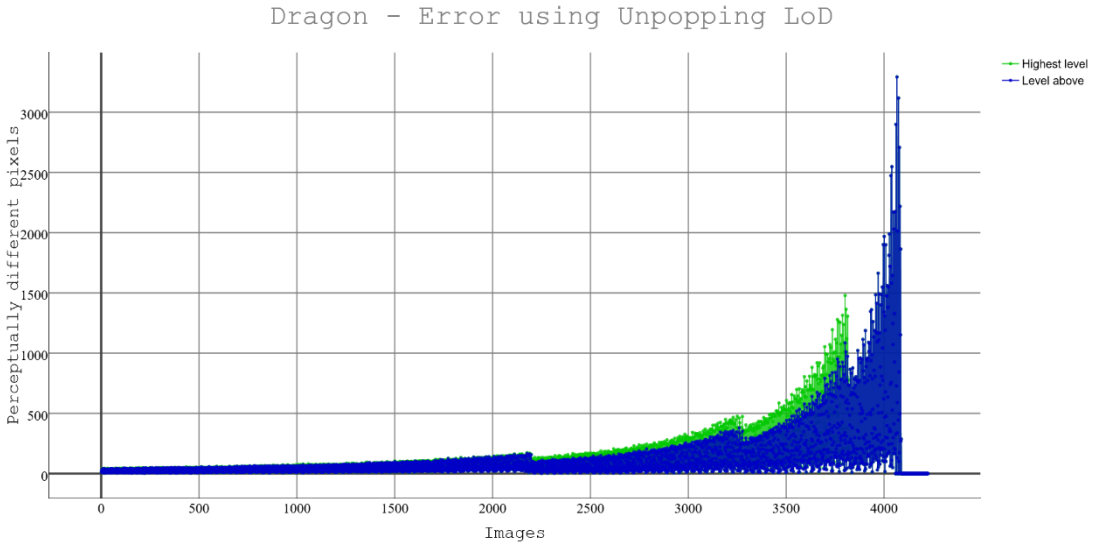

Figure 27. Continuous error of the Dragon using Unpopping LoD

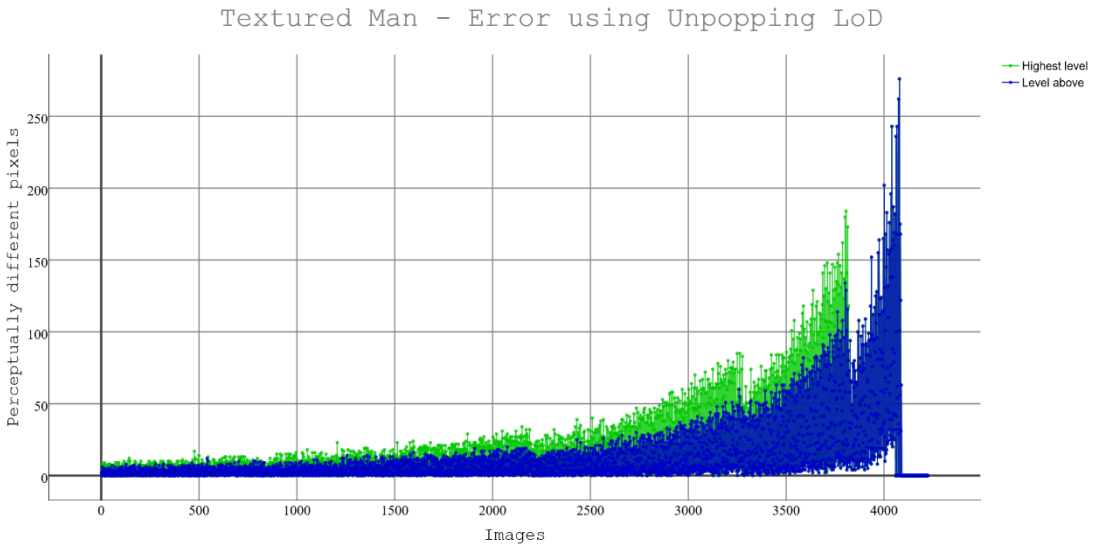

Figure 28. Continuous error of the Textured Man using Unpopping LoD

Unpopping LoD shows basically the same results as Static LoD because they both use the static base models without any improvement or refinement.

5.1.2.3 Curved PN Triangles

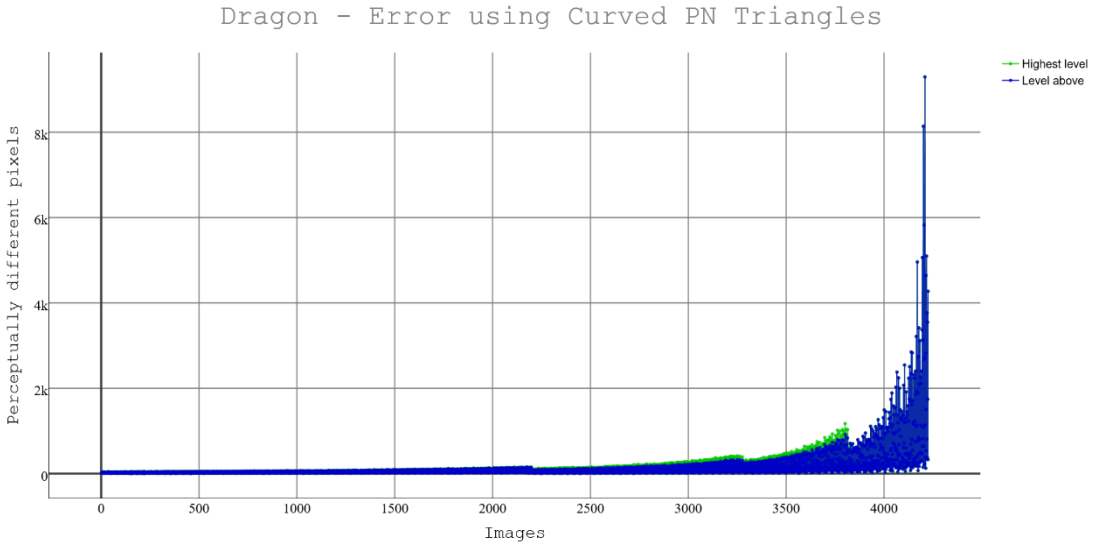

Figure 29. Continuous error of the Dragon using Curved PN Triangles

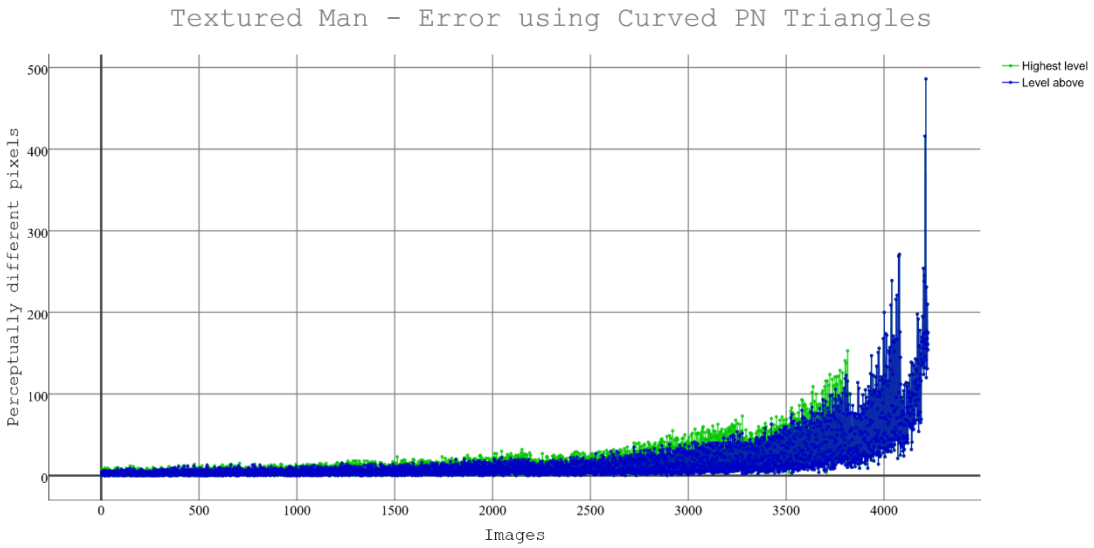

Figure 30. Continuous error of the Textured Man using Curved PN Triangles

The error for this technique shows the same pattern as both Static LoD and Unpopping LoD, but there is a big difference at the end. As this technique generates more triangles and smooths the shapes, the mesh used is not matched to the mesh that it is compared to. As the reference mesh is the goal, this means that towards the end this technique improves too much, and starts going beyond the goal.

5.1.2.4 Phong Tessellation

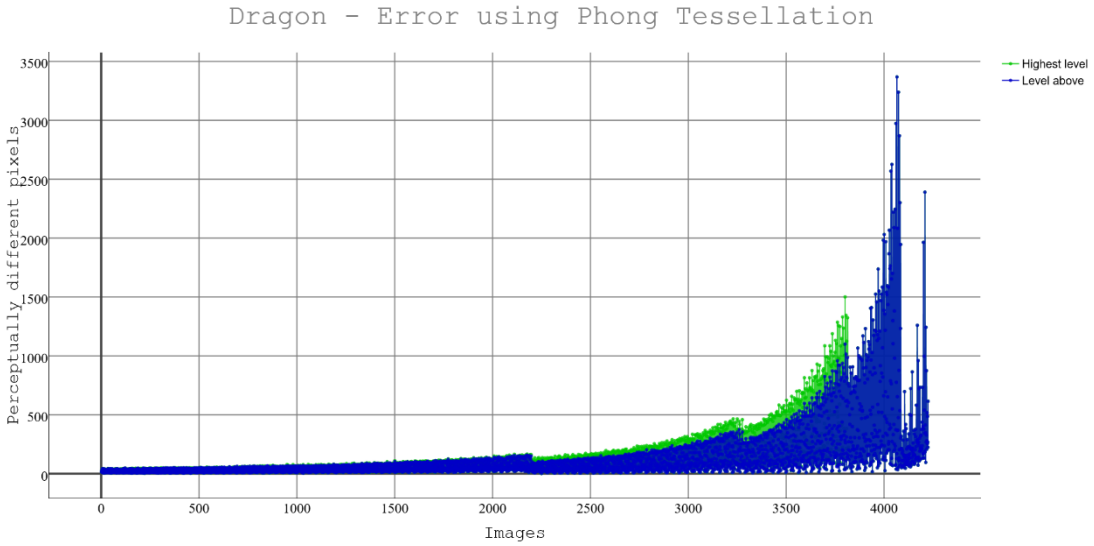

Figure 31. Continuous error of the Dragon using Phong Tessellation

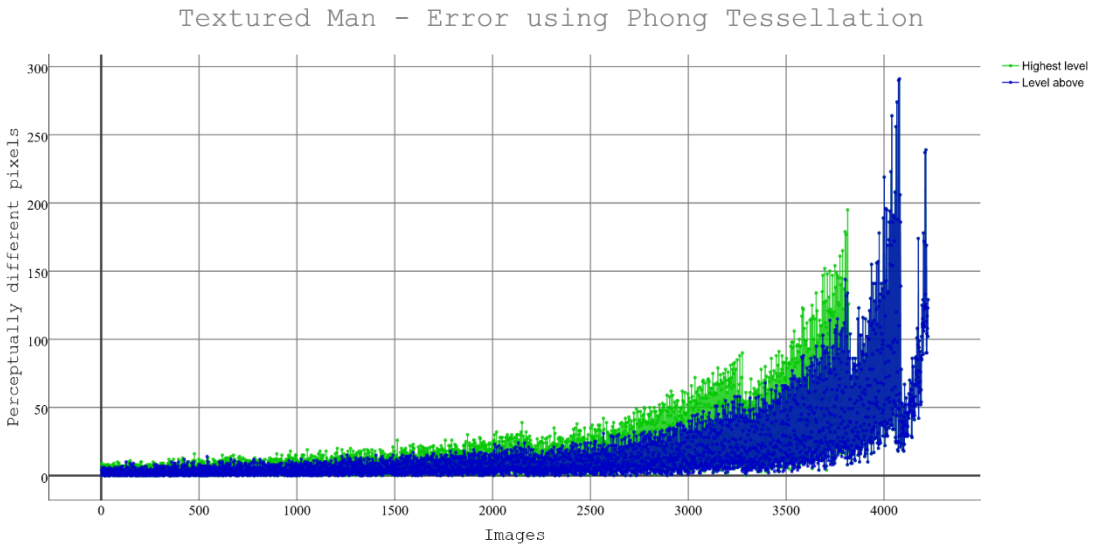

Figure 32. Continuous error of the Textured Man using Phong Tessellation

Again, the same pattern is shown in the results for this technique as well. It also differs from Static LoD and Unpopping LoD towards the end, as Curved PN Triangles does. However, there is a big difference compared to Curved PN Triangles in that the error for this technique is not nearly as high, which means it is closer to the goal of generating geometry matching the reference mesh.

5.1.2.5 Worst Case Comparison

The graphs above contain the results of the analysis of the whole datasets. As the results were very close, this section presents a summary of the graphs where every setup has its own graph with all four techniques included to be more easily comparable. For example, using the Dragon and comparing to the highest level can be seen in Figure 34 below.

To create these graphs the data was divided into 65 groups of 65 elements, then the highest error in every group was included in the graph. The highest was used because the aim is to have as low error as possible, and then it is of higher value to know the worst case scenario. The groups are the X-axis in the graphs, and the highest error of the groups are on the Y-axis. For example, 5 on the X-axis represents group 5, which consist of images 325-390, and the highest error of these 65 images are the value on the Y-axis.

The error for Static LoD and Unpopping LoD is more or less the same, which is why only the line for Unpopping LoD is visible. The line for Static LoD is beneath it.

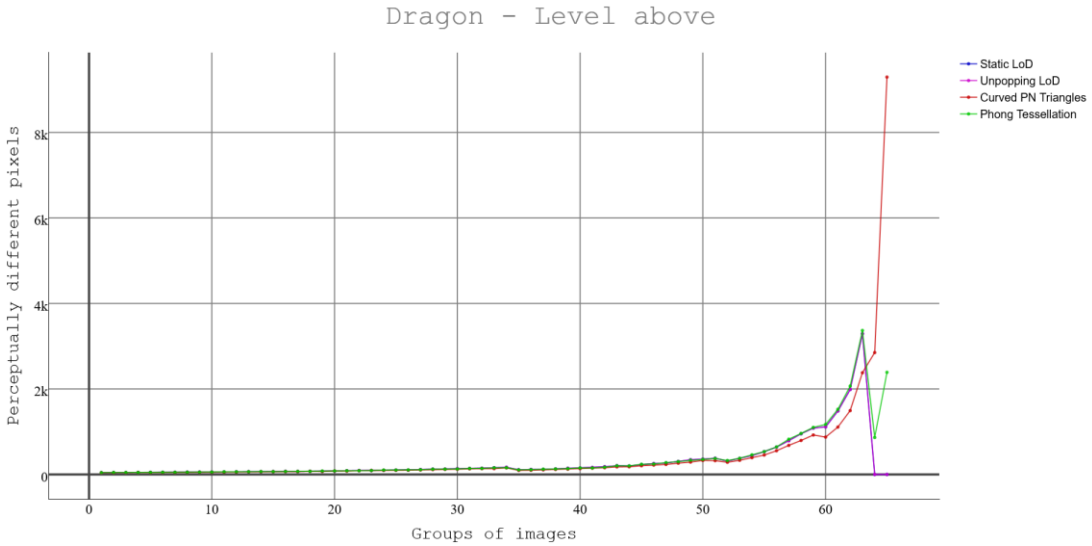

Figure 33. Summary of the continuous error with the Dragon compared to the level above in quality.

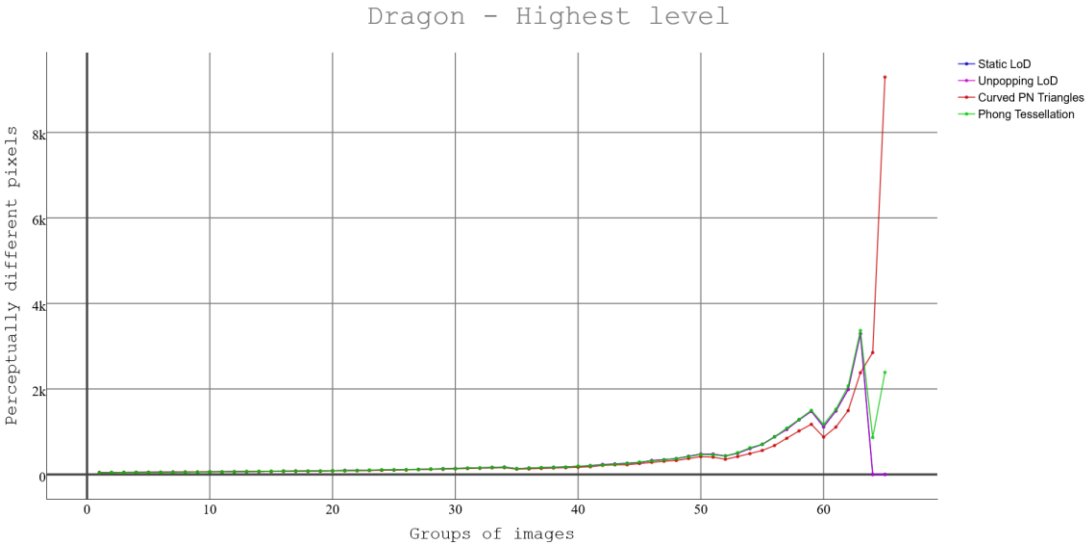

Figure 34. Summary of the continuous error with the Dragon compared to the highest level of quality.

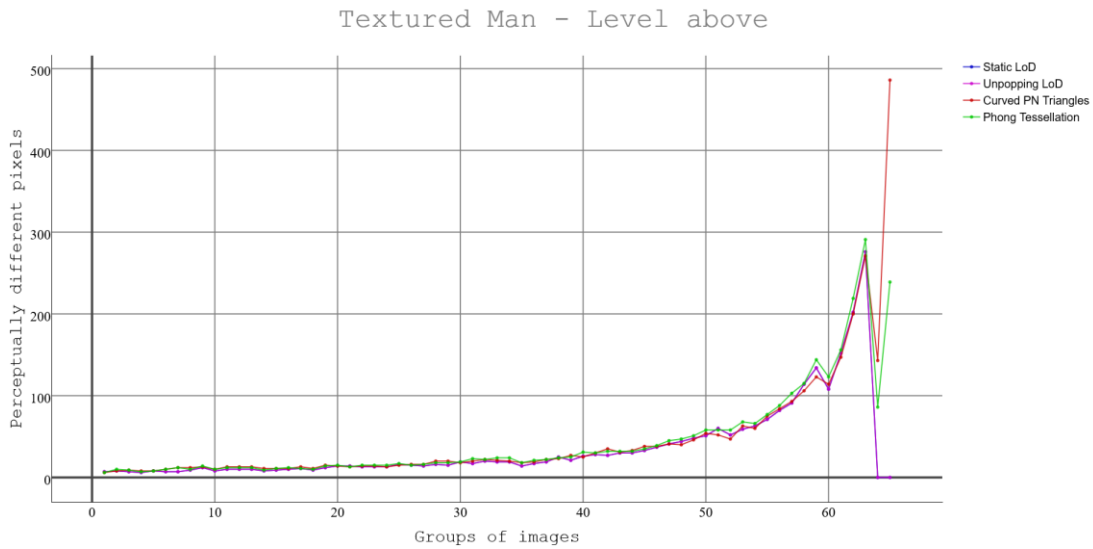

Figure 35. Summary of the continuous error with the Textured Man compared to the level above in quality.

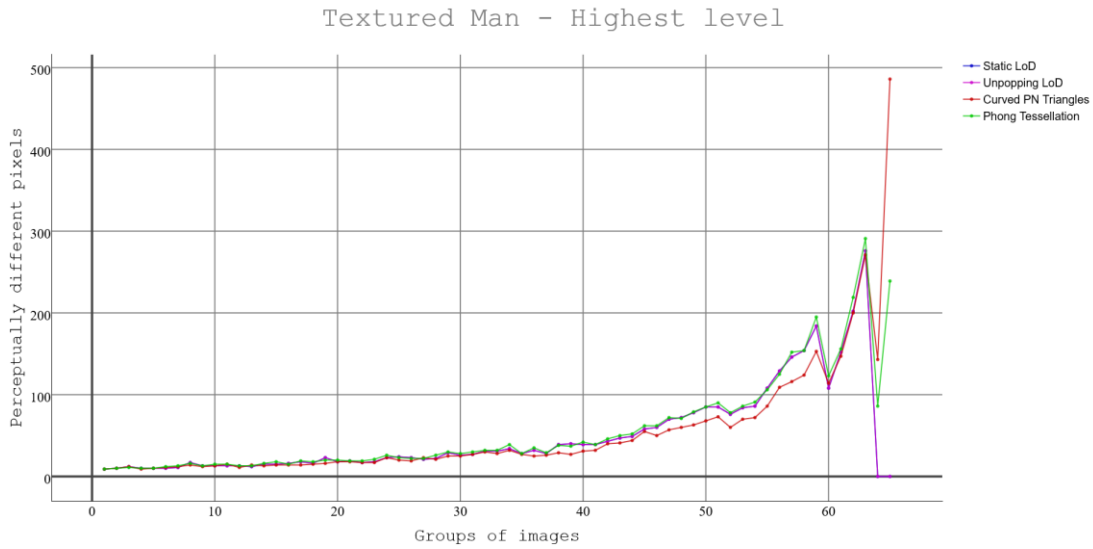

Figure 36. Summary of the continuous error with the Textured Man compared to the highest level of quality.

In these graphs the differences over the whole sequence is clearer than the graphs using the whole dataset. During most of the sequence, Phong Tessellation performs about the same as Static LoD and Unpopping LoD, except for the end where it has much higher errors. Curved PN Triangles performs with a lower error than the other three for a big part of the sequence, which means it is closest to the goal of looking like the reference image. However, towards the end the error is much higher than the other three.

5.1.3 Switching Error/Popping

These experiments were performed a little differently than the two previous ones. To be able to capture the popping for all four techniques, the camera starts at a distance beyond the switch range furthest away. It then moves toward the object without rotating around it, and when a switch is about to happen it captures the frame just before, and then the frame just after. For Unpopping LoD it detects a switch and then captures all frames during the blend-interval.

When a new sequence is started it will use a modified rotation, making the camera view it from a difference angle. This is important because the amount of popping depends on the angle from which the object is seen. Looking at the man from above generates a much lower error that if it is seen from the front. A total of 80 sequences were sampled for each technique, and all of them were used to calculate the averages in Figure 45 and Figure 46.

The amount of error differs between the two objects for the same reason as the continuous error, which is shown in Figure 13. The Dragon has more detail over the whole mesh than the man, creating errors all over the mesh. The Man has less shapes and the majority of the errors will be found on the silhouette.

The green lines named Run in the charts of Static LoD, Phong Tessellation, and Curved PN Triangles is representing one sequence where the camera started from far away and then moved towards the object. The blue lines named Avg is the average of all measured switches on that level.

For the Unpopping LoD results to be a fair comparison to the other techniques the chart shows Avg, Min, and Max error for every switch. These are represented by the blue, green, and red lines respectively.

5.1.3.1 Static LoD

Figure 37. Error caused by switching (popping) with the Dragon using Static LoD.

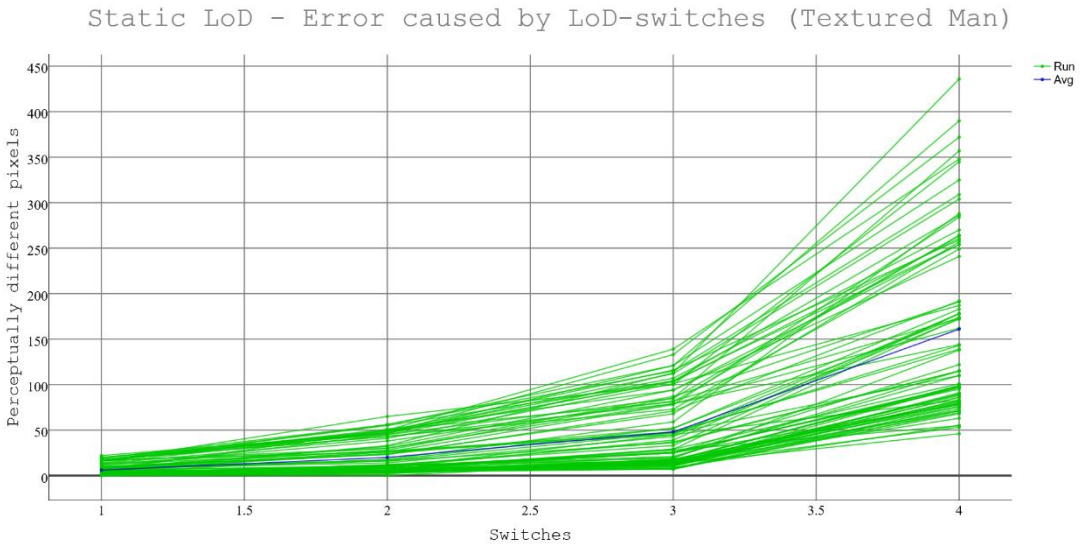

Figure 38. Error caused by switching (popping) with the Textured Man using Static LoD.

The pattern for the switches is similar for both of the models, especially looking at the average case. However, as stated in section 5.1.3, the amount of error in number of pixels is much higher using the Dragon than when using the Man.

5.1.3.2 Unpopping LoD

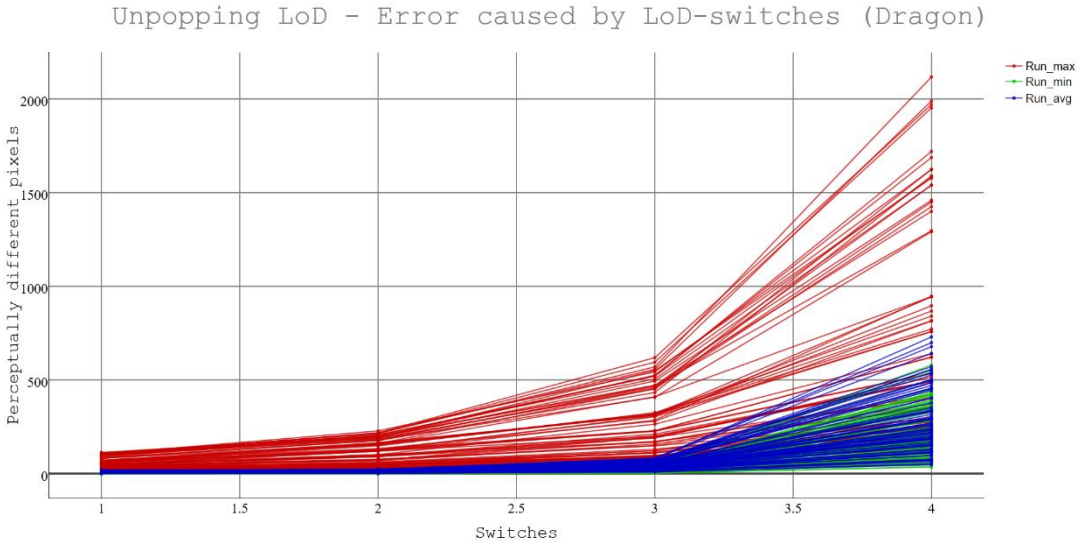

Figure 39. Error caused by switching (popping) with the Dragon using Unpopping LoD.

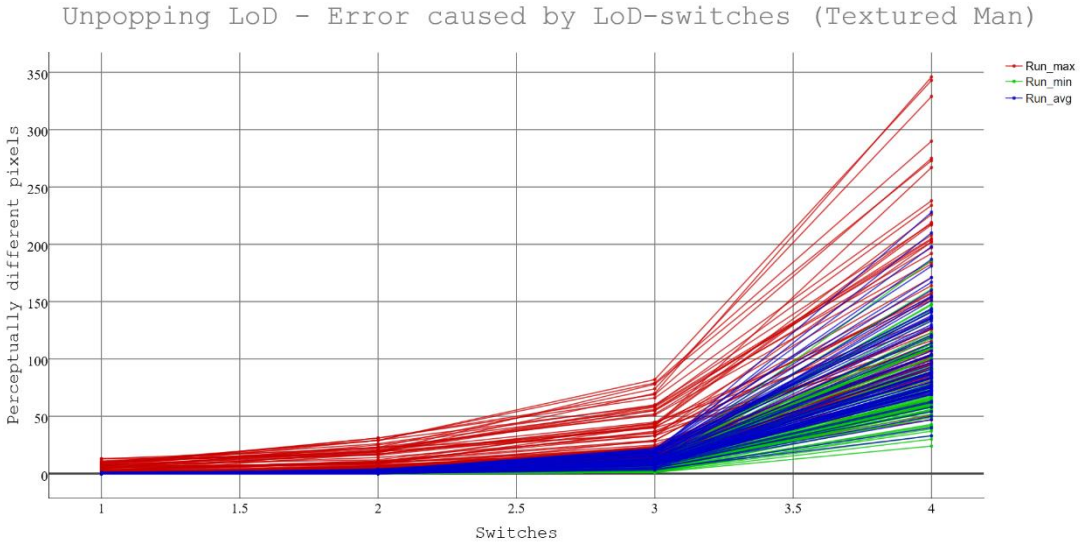

Figure 40. Error caused by switching (popping) with the Textured Man using Unpopping LoD.

The error for Unpopping LoD is measured using all the frames during the blending interval, comparing the first to the second, the second to the third, the third to the fourth, and so on. The maximum, minimum, and average is calculated, which is represented by the red, green, and blue lines respectively.

The pattern is similar to Static LoD, but Unpopping LoD shows much lower errors. This means that during the blending interval there are visible changes, but since it is spread over several frames the switch is much less visible.

5.1.3.3 Curved PN Triangles

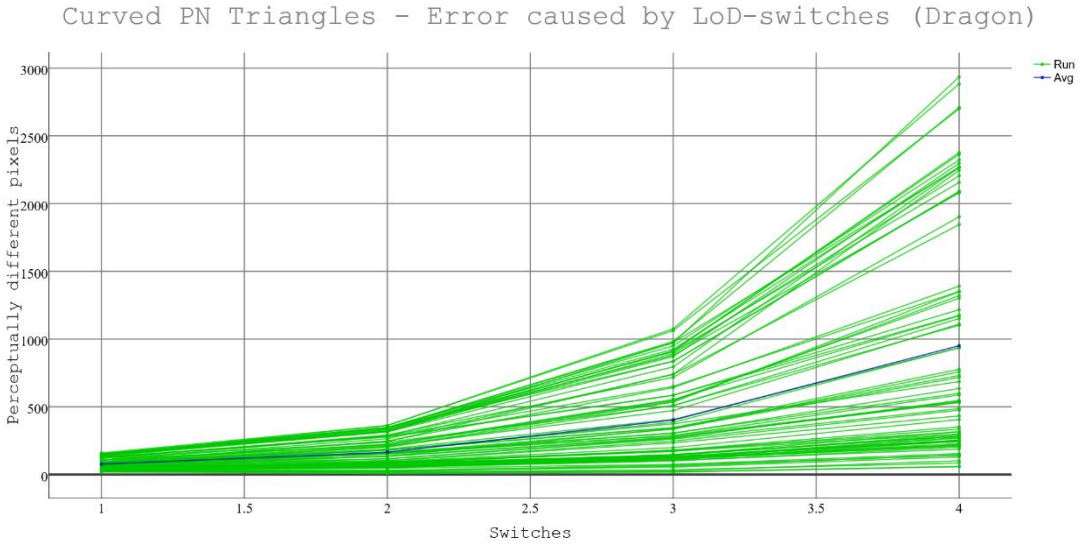

Figure 41. Error caused by switching (popping) with the Dragon using Curved PN Triangles.

Figure 42. Error caused by switching (popping) with the Textured Man using Curved PN Triangles.

This technique does not show as low results as Unpopping LoD, but they are much lower than Static LoD. This means that the increased detail of the mesh matches more closely to the mesh after the switch, which means the pop is less visible.

5.1.3.4 Phong Tessellation

Figure 43. Error caused by switching (popping) with the Dragon using Phong Tessellation.

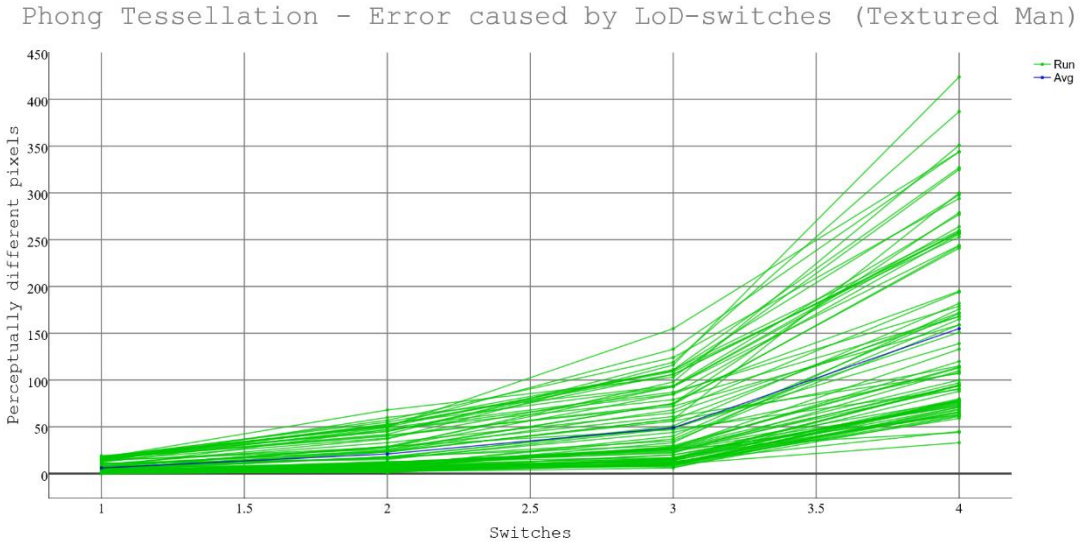

Figure 44. Error caused by switching (popping) with the Textured Man using Phong Tessellation.

Phong Tessellation has a slightly higher average error compared to Curved PN Triangles, and some runs are similar to those of Static LoD. However, the average is lower than Static LoD, which means that the popping is less visible. However, it is slightly more visible than Curved PN Triangles.

5.1.3.5 Averages Compared

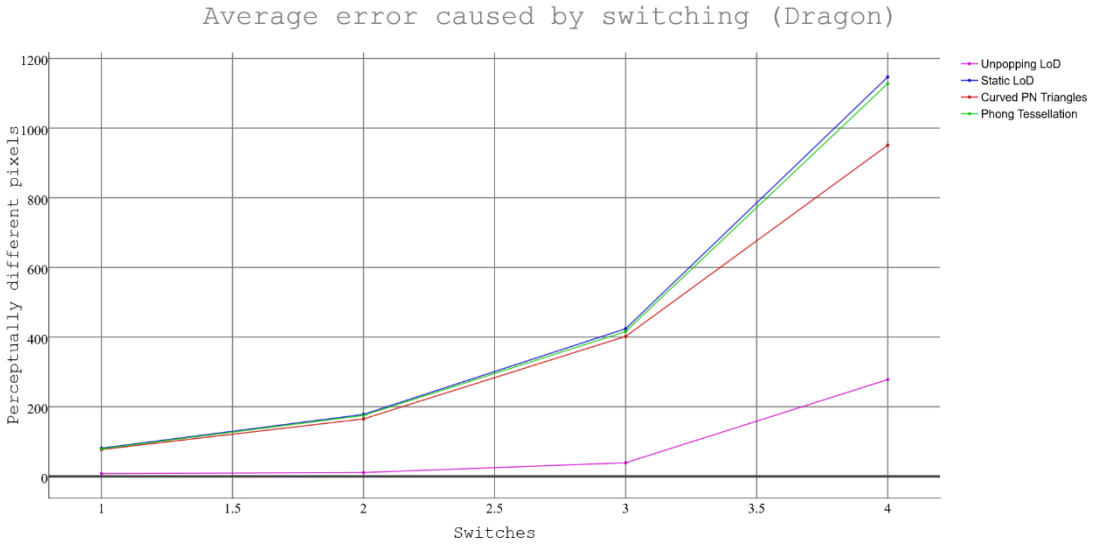

Figure 45. The average errors of all four techniques caused by the switching of meshes of the Dragon.

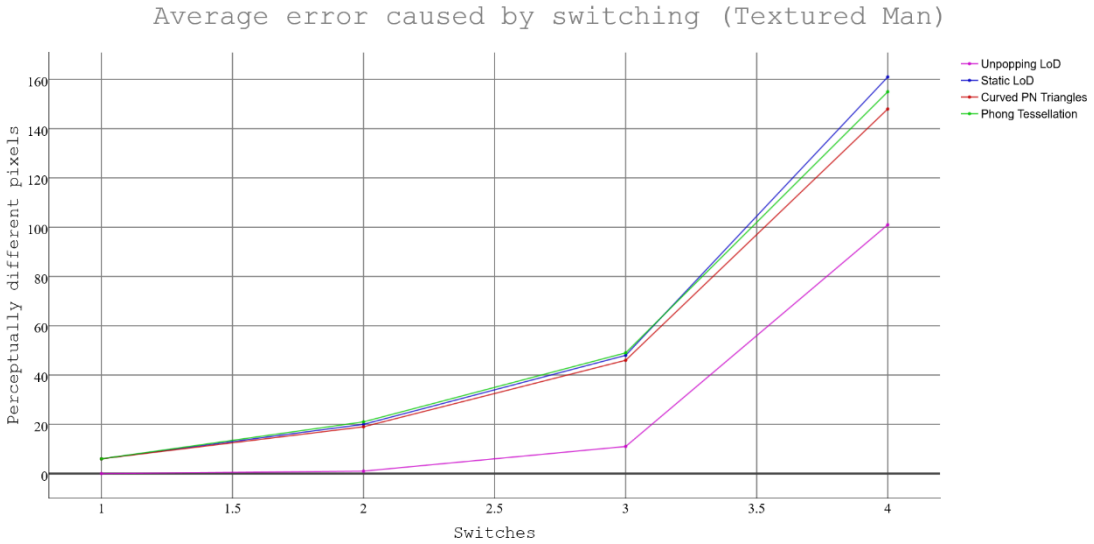

Figure 46. The average errors of all four techniques caused by the switching of meshes of the Textured Man.

These two charts show the averages of all four techniques compared for both models, and the results are very clear. The average of Unpopping LoD is much lower than the other three techniques, which show similar results compared to each other. The error of Curved PN Triangles is slightly lower than both Phong Tessellation and Static LoD overall. On the last switch of the Dragon it is much lower, but since this is not the case with the Man it may just depend on the model used.

Static LoD is the worst with both models, and Phong Tessellation is only slightly better.

5.1.4 Results Summarized

This section summarizes the results of the experiments. The FPS, number of triangles, and popping are ordered according to the data from the experiments. The continuous error is a little harder to order because Curved PN Triangles had better results than the other three techniques during parts of the sequences, but much worse during other parts. During the parts where the error of Curved PN Triangles was higher than the other three techniques it was so high that these parts were considered more important than the parts where it was lower than the other three techniques. This is why it is on the fourth place.

	FPS (highest to lowest)	**Number of triangles (highest to lowest)**
1	Static LoD	Curved PN Triangles
2	Unpopping LoD	Phong Tessellation
3	Phong Tessellation	Unpopping LoD
4	Curved PN Triangles	Static LoD

Table 5. Summary of the performance results.

	Continuous error (lowest to highest)	**Popping average (lowest to highest)**
1	Static LoD	Unpopping LoD
2	Unpopping LoD	Curved PN Triangles
3	Phong Tessellation	Phong Tessellation
4	Curved PN Triangles	Static LoD

Table 6. Summary of the image space results.

5.2 Discussion

The best result of the dynamic techniques is when the meshes are of very low quality, because smoothing them creates a much bigger difference than smoothing an object that already has a high amount of detail. The problem with this is that the low detail meshes are used at such a long distance that the smoothing is not necessarily that noticeable, and when the object is closer it uses a higher detailed mesh that already looks good. This kind of defeats the purpose of the dynamic techniques. There would probably be a much better result if all base meshes were of lower quality. For example, the highest level of the Textured Man used in these experiments consists of 50,000 triangles. If it instead would consist of only 5,000 triangles the smoothing would probably be much more needed, and the silhouette-focus of Phong Tessellation would be worth the extra resources it uses.

Tessellation creates a better result when the factors can go higher, because the higher the range of factors used, the more intermediate meshes can be created. But with high factors it also means that there will be very much geometry created, which leads to lower performance. In these experiments, when using the mesh with the highest level, the triangle count was as high as 500,000, which is often unnecessarily high. This could also be improved by using base meshes of lower detail.

This big difference in triangle count may seem unfair, but to be able to compare the dynamic techniques against the ones using static meshes they need to have a common ground to start with. This would probably be the case for a developer who is comparing the techniques. The first step would not be to create several different sets of meshes to use with the different techniques, because then the results can be influenced by specific characteristics of the set of meshes the techniques are using. Using the same meshes prevents this, and provides a comparison of only the techniques and their functionality. When the comparison is done, depending on the results, optimizations could be made to provide a better result for the dynamic techniques.

Other important factors to take into considerations are the prerequisites of the techniques. The simplicity of Static LoD makes it possible to use in all kinds of applications that can render graphics. Unpopping LoD needs to utilize the blending functionality of the graphics hardware, which could cause problems when using other rendering techniques. For example, using blending together with deferred rendering can often be tricky.

Both Phong Tessellation and Curved PN Triangles rely on the tessellation unit, which is the latest addition to the pipeline. If the game or game engine has not been updated to use this latest version it could mean that a major update is necessary to be able to use tessellation. They will also not work on hardware that does not support tessellation.

Due to time constraints, only a subset of all rendered frames could be examined. If, for example, the object was to be viewed from every possible direction, using whole degrees for the angles, at a range decrease of 1 distance unit per full rotation, it would result in tens of millions of images. With the limited time and hardware for this study, handling that many images would not be feasible. However, enough images were analyzed to show clear results and patterns in both of the experiments and for all four techniques. In total, just over 105,000 images were analyzed.

Because of the time constraints, the tests were also performed on a single system. Performing the tests on several systems using different hardware could show different results, but as the hardware used is relatively up-to-date in the sense that it, for example, uses the latest version of Direct3D, it would probably show the same conclusion. The numbers may be different, but the relative difference between the techniques would most likely be similar, which is what is tested in this study.

Conducting experiments of perceptual nature can be a hard task. The human visual system is advanced, and it is hard to conclude if something will be noticed or not. If the

user is looking at the exact position at the exact moment a change happens, it might be enough with an error of a single pixel to be noticed. Other times there may be many erroneous pixels, but the user might not notice them because his focus is elsewhere. However, a lower error is always good.

The experiments in this study focused on the difference between the techniques, using the same tool to measure all the errors. This means that the importance of the exact values of the errors becomes lower, and the importance of the differences between the rrors becomes higher. These differences are often in the form of patterns or observations such as “technique A has generally lower error than technique B”, which makes it easier to draw conclusions. It also puts less demands on the model used to represent the human visual system, even if it is still very important.

6 CONCLUSION

The dynamic techniques did show some unexpected results in that they did not perform as well as expected. Their quality is better than Static LoD, but with much lower FPS. However, if any of these two techniques should be used, Phong tessellation is the one to choose because it has very similar visual results compared to Curved PN Triangles while having less extra geometry and much better FPS.

These techniques should also be used with objects consisting of round shapes due to the deformation it causes when smoothing objects with hard edges and corners. The objects themselves should preferably also be adapted to the use of the dynamic techniques by, for example, having a more suitable amount of geometry to start with, where the silhouette-focus can be utilized better to create a bigger difference.

From the results presented in section 5.1, the single best of the four techniques used in this study is Unpopping LoD. It has good performance and provides a good visual result with the lowest average amount of popping of the compared techniques.

The dynamic techniques are not suitable as a substitute to Unpopping LoD, but further research could be conducted to examine how they can be used together.

7 FUTURE WORK

This research has provided an insight to how the techniques work and the comparison of the different results interpreted by a computer. The model used to compare the images tries to emulate the human visual system as close as possible, but it is still only a model. To provide even more conclusive results, experiments should be conducted with human participants to see how the real human visual system will perceive the differences between the techniques.

Further experiments should also be conducted concerning the results in complex 3D environments which is where the objects will most often be used. The possibility of combining the techniques should also be examined. For example, using Static LoD for the level furthest away where it might not be noticeable enough to use the extra resources of Unpopping LoD. Then use Unpopping LoD for the levels closer to the object, where spending the extra resources would give a higher value in that it leads to a better visual result. This could also be combined with Phong Tessellation and the use of lower detailed meshes.

Another thing that should be examined in this context is how to use tessellation together with displacement mapping to generate more detail of a mesh, instead of using the dynamic techniques presented here. Since the object would have data specifying where and how much to tessellate, it gives the designers more control compared to, for example, the brute force smoothing of Curved PN Triangles.

REFERENCES

[1] J. Clark. “Hierarchical Geometric Models for Visible Surface Algorithms”. *Communications of the ACM*, October 1976 Volume 19 Number 10. Pages 547-554. University of California at Santa Cruz.

[2] D. P. Luebke, *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2003. ISBN: 978-1-55860-838-2

[3] M. Giegl and M. Wimmer, “Unpopping: Solving the Image-Space Blend Problem for Smooth Static LOD Transitions”, in *Computer Graphics Forum*, 2007, vol. 26, pp. 46–49.

[4] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell, “Curved PN triangles”, in *Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001, pp. 159–166.

[5] T. Boubekeur and M. Alexa, “Phong tessellation”, *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5, p. 141, 2008.

[6] N. Dyn, D. Levine, and J. A. Gregory, “A butterfly subdivision scheme for surface interpolation with tension control”, *ACM transactions on Graphics (TOG)*, vol. 9, no. 2, pp. 160–169, 1990.

[7] E. Catmull, and J. Clark. 1978. “Recursively Generated B-Spline Surfaces on Arbitrary Topology Meshes.” *Computer Aided Design* 10(6), pp. 350–355.

[8] L. Kobbelt, “√ 3-subdivision”, in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 103–112.

[9] C. Loop, “Smooth Subdivision Surfaces Based on Triangles”, University of Utah, 1987.

[10] T. Boubekeur, P. Reuter, and C. Schlick, “Scalar tagged PN triangles”, in *EUROGRAPHICS Short Papers*, 2005.

[11] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, “A developer’s guide to silhouette algorithms for polygonal models”, *Computer Graphics and Applications, IEEE*, vol. 23, no. 4, pp. 28–37, 2003.

[12] J. M. Beusmans, D. D. Hoffman, and B. M. Bennett, “Description of solid shape and its inference from occluding contours”, *JOSA A*, vol. 4, no. 7, pp. 1155–1167, 1987.

[13] W. Thompson, R. Fleming, S. Creem-Regehr, and J. K. Stefanucci, “Visual perception from a computer graphics perspective”, CRC Press, 2011.

[14] H. Yee, “Perceptual Metric for Production Testing”, *Journal of Graphics Tools*, vol. 9, no. 4, pp. 33–40, Jan. 2004.

[15] Crytek GmbH. “Creating LODs - CRYENGINE Manual – Documentation”. [Online]. Available: http://docs.cryengine.com/display/SDKDOC2/Creating+LODs. [Accessed: 27-Apr-2016].

[16] Interactive Data Visualization, Inc. “lod [SpeedTree Documentation]”. [Online]. Available: http://docs.speedtree.com/doku.php?id=lod. [Accessed: 27-Apr-2016].

[17] Valve, “ValveSoftware/source-sdk-2013”, *GitHub*. [Online]. Available: https://github.com/ValveSoftware/source-sdk-2013. [Accessed: 27-Apr-2016]. File: studio.h

[18] “LOD - Bohemia Interactive Community”. [Online]. Available: https://community.bistudio.com/wiki/LOD. [Accessed: 27-Apr-2016].

[19] Epic Games, “Performance Guidelines for Artists and Designers”. [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/Performance/Guidelines/. [Accessed: 02-May-2016].

[20] “Ryse Polygon Count Comparison with Other AAA Titles - Star Citizen, Crysis 3 and More”, *WCCFtech*, 28-Sep-2013.

[21] Unity Technologies “Unity - Manual: Modeling Characters for Optimal Performance”. [Online]. Available: http://docs.unity3d.com/Manual/ModelingOptimizedCharacters.html. [Accessed: 02-May-2016].

[22] G. Nelva, ‘The Numbers of Killzone: Shadow Fall Revealed: 40,000 Polygons per Character, 683,334 Building Blocks and More | DualShockers’. [Online]. Available: http://www.dualshockers.com/2014/03/23/the-numbers-of-killzone-shadow-fall-revealed-40000-polygons-per-character-683334-building-blocks-and-more/ [Accessed: 04-May-2016].

[23] G. Nelva, ‘Xbox One Exclusive Sunset Overdrive Has 40,000 Polygons Per Character and Physically Simulated Cloth | DualShockers’. [Online]. Available: http://www.dualshockers.com/2014/05/08/xbox-one-exclusive-sunset-overdrive-has-40000-polygons-per-character-and-physically-simulated-cloth/ [Accessed: 04-May-2016].

[24] G. Nelva, ‘The Order: 1886 Characters have over 100,000 Polygons, More than Ryse’s Marius, but Less Blend Shapes | DualShockers’. [Online]. Available: http://www.dualshockers.com/2014/02/20/the-order-1886-characters-have-over-100k-polygons-more-than-ryses-marius-but-less-blend-shapes/ [Accessed: 04-May-2016].

[25] ‘The Stanford 3D Scanning Repository’. [Online]. Available: http://graphics.stanford.edu/data/3Dscanrep/. [Accessed: 05-May-2016].

[26] ‘3d 3ds industrial truck’. [Online]. Available: http://www.turbosquid.com/3d-models/3d-3ds-industrial-truck/249922. [Accessed: 05-May-2016].

[27] ‘Free 3D Models of Humans’, *Renderpeople*, 06-Oct-2015. [Online]. Available: https://renderpeople.com/free-3d-model/. [Accessed: 09-May-2016].

[28] ‘Rendering Cubic Bezier Patches’. [Online]. Available: https://web.cs.wpi.edu/~matt/courses/cs563/talks/surface/bez_surf.html. [Accessed: 17-May-2016].

[29] E. W. Weisstein, ‘Spherical Coordinates’. MathWorld--A Wolfram Web Resource [Online]. Available: http://mathworld.wolfram.com/SphericalCoordinates.html. [Accessed: 17-May-2016].

[30] “Graphics Pipeline (Windows)”. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx. [Accessed: 02-May-2016]

[31] “Tessellation Overview (Windows)”. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx. [Accessed: 02-May-2016]

[32] Crytek, ‘Tessellation and Displacement - CRYENGINE Manual - Documentation’. [Online]. Available: http://docs.cryengine.com/display/SDKDOC2/Tessellation+and+Displacement. [Accessed: 07-Jun-2016].

[33] ‘Metro 2033. | Description | GeForce’. [Online]. Available: http://www.geforce.com/games-applications/pc-games/metro-2033/description. [Accessed: 07-Jun-2016].

[34] Epic Games, Inc, ‘1.8 - Tessellation’. [Online]. Available: https://docs.unrealengine.com/latest/INT/Resources/ContentExamples/MaterialProperties/1_8/. [Accessed: 07-Jun-2016].

[35] Unity Technologies, ‘Unity - Manual: Surface Shaders with DX11 / OpenGL Core Tessellation’. [Online]. Available: http://docs.unity3d.com/Manual/SL-SurfaceShaderTessellation.html. [Accessed: 07-Jun-2016].

[36] Pixar, ‘OpenSubdiv::OPENSUBDIV_VERSION::Sdc Namespace Reference’. [Online]. Available: http://graphics.pixar.com/opensubdiv/docs/doxy_html/a00163.html#aa3daa2f428c6be2b1cb394f3a525833d. [Accessed: 07-Jun-2016].

APPENDIX A – CURVED PN TRIANGLES CODE

Hull Shader

This code snipped is constructing the patch constants used in the Domain Shader. It is implemented according to the details specified in [4].

```
//Assign Positions and normals
float3 B003 = inputPatch[0].pos.xyz;
float3 B030 = inputPatch[1].pos.xyz;
float3 B300 = inputPatch[2].pos.xyz;

float3 N002 = inputPatch[0].normal;
float3 N020 = inputPatch[1].normal;
float3 N200 = inputPatch[2].normal;

//Compute the cubic geometry control points
//Edge control points
output.B210 = ((2.0f * B003) + B030 - (dot((B030 - B003), N002) * N002)) / 3.0f;
output.B120 = ((2.0f * B030) + B003 - (dot((B003 - B030), N020) * N020)) / 3.0f;
output.B021 = ((2.0f * B030) + B300 - (dot((B300 - B030), N020) * N020)) / 3.0f;
output.B012 = ((2.0f * B300) + B030 - (dot((B030 - B300), N200) * N200)) / 3.0f;
output.B102 = ((2.0f * B300) + B003 - (dot((B003 - B300), N200) * N200)) / 3.0f;
output.B201 = ((2.0f * B003) + B300 - (dot((B300 - B003), N002) * N002)) / 3.0f;

//Center control point
float3 E = (output.B210 + output.B120 + output.B021 + output.B012 + output.B102 +
output.B201) / 6.0f;
float3 V = (B003 + B030 + B300) / 3.0f;
output.B111 = E + ((E - V) / 2.0f);

// Compute the quadratic normal control points, and rotate into world space
float V12 = 2.0f * dot(B030 - B003, N002 + N020) / dot(B030 - B003, B030 - B003);
output.N110 = normalize(N002 + N020 - V12 * (B030 - B003));
float V23 = 2.0f * dot(B300 - B030, N020 + N200) / dot(B300 - B030, B300 - B030);
output.N011 = normalize(N020 + N200 - V23 * (B300 - B030));
float V31 = 2.0f * dot(B003 - B300, N200 + N002) / dot(B003 - B300, B003 - B300);
output.N101 = normalize(N200 + N002 - V31 * (B003 - B300));

return output;
```

Domain Shader

This code snippet is calculating the positions, normals, and UV-coordinates of the new geometry. It is implemented according to the details specified in [4].

```
//The barycentric coordinates
float u = domainLocation.x;
float v = domainLocation.y;
float w = domainLocation.z;

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

//Precompute squares * 3
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 =
            patch[0].pos.xyz * ww * w +
            patch[1].pos.xyz * uu * u +
            patch[2].pos.xyz * vv * v +
            hsConstData.B210 * ww3 * u +
            hsConstData.B120 * w * uu3 +
            hsConstData.B201 * ww3 * v +
            hsConstData.B021 * uu3 * v +
            hsConstData.B102 * w * vv3 +
            hsConstData.B012 * u * vv3 +
            hsConstData.B111 * 6.0f * w * u * v;

//Compute normal from quadratic control points and barycentric coords
float3 normal =
            patch[0].normal * ww +
            patch[1].normal * uu +
            patch[2].normal * vv +
            hsConstData.N110 * w * u +
            hsConstData.N011 * u * v +
            hsConstData.N101 * w * v;

normal = normalize(normal);

//Transform position into clip-space
float4 p = mul(float4(position, 1.0f), viewMatrix);
p = mul(p, projectionMatrix);
output.pos = p;

//Linearly interpolate the texture coords
output.uv = patch[0].uv * w + patch[1].uv * u + patch[2].uv * v;

output.normal = normal;

return output;
```

APPENDIX B – PHONG TESSELLATION CODE

Hull Shader

This function is used in the Hull Shader to determine if the input triangle is positioned on the silhouette. If it is, it receives more tessellation than if it is not.

```
/* Returns the orientation adaptive tessellation factor (0.0f -> 1.0f)
edgeDotProduct    - Dot product of edge normal with view vector
silhouetteEpsilon - Epsilon to determine the range of values considered to be
silhoutte
*/
float GetOrientationAdaptiveScaleFactor(float edgeDotProduct, float silhouetteEpsilon)
{
     float scale = 1.0f - abs(edgeDotProduct);
     scale = saturate((scale - silhouetteEpsilon) / (1.0f-silhouetteEpsilon));
     return scale;
}
```

Domain Shader

This code snippet is calculating the positioning and normals etc. of the new geometry. It is implemented according to the details specified in [5].

```
//The barycentric coordinates
float u = domainLocation.x;
float v = domainLocation.y;
float w = domainLocation.z;

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

//Compute position
float3 position =
            patch[0].pos.xyz * ww +
            patch[1].pos.xyz * uu +
            patch[2].pos.xyz * vv +
            w * u * (PI(patch[0], patch[1]) + PI(patch[1], patch[0])) +
            u * v * (PI(patch[1], patch[2]) + PI(patch[2], patch[1])) +
            v * w * (PI(patch[2], patch[0]) + PI(patch[0], patch[2]));

float t = 0.5;
position = position * t + (patch[0].pos.xyz * w + patch[1].pos.xyz * u +
patch[2].pos.xyz * v)*(1 - t);

//Compute normal
float3 normal =
            patch[0].normal * w +
            patch[1].normal * u +
            patch[2].normal * v;

normal = normalize(normal);

//Transform position into clip-space
float4 p = mul(float4(position, 1.0f), viewMatrix);
p = mul(p, projectionMatrix);
output.pos = p;

//Linearly interpolate the texture coords
output.uv = patch[0].uv * w + patch[1].uv * u + patch[2].uv * v;

output.normal = normal;

return output;
```