Master in Computing
Llenguatges i Sistemes Informàtics

Master Thesis
– September 2010 –

# RENDERING CURVED TRIANGLES ON THE GPU

Student: *Rodrigo Pizarro Lozano*
Advisor/Director: *Antonio Chica*

UNIVERSITAT POLITÈCNICA DE CATALUNYA

|  |  |
|---|---|
| **Title:** | Rendering Curved Triangles on the GPU |
| **Author:** | Rodrigo Pizarro Lozano |

|  |  |
|---|---|
| **Advisor/Director:** | Antonio Chica |
| **Date:** | September 2010 |

**Abstract:**

This Thesis presents a new approach to render triangular Bézier patches in real time. The goal is to achieve a very good visual quality, avoid artifacts in the silhouette, and get infinite detail.

Our approach consists in a ray casting technique to render triangular Bézier patches in real time. It is based on previous work explained in this document to implement a fast ray-surface intersection technique. This previous work consists in adapting Newton's method to implement the intersections achieving interactive framerates ray casting different surfaces.

The main contributions of our approach are adapting Newton's method to perform intersections with triangular bicubic Bézier patches and implementing it in GPU to optimize performance using graphics hardware.

Finally, we also contribute adapting the normal mapping technique to shade the models and, thus, achieve even greater detail.

**Keywords:** PN surface, Bézier, Level of detail, Ray casting, GPU
**Language:** English
**Modality:** Research Work

## Acknowledgments

I would like to thank my Thesis tutor for his dedication and patience throughout all the development of this document and implementation.

I also would like to thank specially David Aguilera and Toni Villegas for all the years we have been working together, and for all the help received in the times when I most needed it, in my professional and personal life.

Thanks to all my teachers, and specially to Juan Eugenio Padilla for everything he has done for me since I can remember, for always being ready for any question I could have.

Thanks to all communities that developed all the libraries I used in this Thesis.

# Contents

CHAPTER 1

# Introduction

This chapter begins with the aim of the Thesis and its research interest. Reading this section is key to understand the motivation and goals of the entire document. Section 1.2 is a short description of the content in all chapters.

## 1.1 Aim of this Thesis

Rendering complex models is still an open issue. The improvements in graphics hardware only cannot deal with the even bigger growth in complexity of the models. Therefore, software solutions are needed to take advantage of the new hardware and optimize the rendering.

One of the most commonly used techniques is LOD (level of detail). LOD-based techniques are a very well studied field, but there are still problems that have yet to be solved. There are many methods to render low resolution models, but they can be classified into two categories:

- Geometrically based methods
- Image or shading based methods

Geometrical methods tend to be intensive hardware using. This usually leads to a lower performance in rendering time or memory use. Image based methods, on the other hand, have a very good performance, but artifacts or non correct shapes might be visible in the silhouette and also zooming in. We here present a real time technique capable of solving those problems.

Our approach consists in a ray casting technique to render triangular Bézier patches in real time. It is based in the articles [2] and [5] to implement a fast ray-surface intersection technique. In these articles they adapt Newton's method to implement the intersections achieving interactive framerates ray casting different surfaces, such as NURBS and square Bézier patches. In order to be able to model any shape, a trimming is needed. Also, the authors do not use any graphic acceleration.

The main contributions of our approach are adapting Newton's method to perform intersections with triangular Bézier patches and taking advantage of graphics hardware.

The surfaces used and its shading is based in the work of Vlachos et. al. in the article [9]. The use of triangular Bézier patches allows modeling any shape without trimming the surfaces, which make them more suitable than square ones. Adapting Newton's method to calculate the intersections between rays and triangular Bézier patches allows very fast computations and its our first major contribution.

Using GPU we achieve a boost in performance, as seen in the results chapter. In order to be able to use it, the rendering technique has to be adapted. This adaptation is the second major contribution in this Thesis.

Finally, we also contribute adapting the normal mapping technique to shade the models and, thus, achieve even greater detail.

## 1.2 Outline of this document

In chapter 2 we describe the state of the art in our area of research in computer graphics. It is divided into two sections. The first one consists in a series of basic concepts. In the second one we describe various articles that contain the theoretical basis used in this Thesis. These In particular, the articles [5], [2] and [9] contain the most part of the previous work in our approach.

Chapter 3 contains the details of the implementation of our approach. In section 3.1 and subsections, we describe in depth our ray tracing technique, which is the central part of this Thesis. In further sections we describe how we perform the shading, the implemented optimizations and the formats supported to read 3D models.

Chapter 4 displays the results of various executions to show the implemented techniques. We focus in the aspects commented in previous chapters. There are three sections, one for each step in the implementation process. The first step contains the results of our approach executing a CPU implementation, whereas the other steps are GPU implementations. A comparison is also displayed.

In the final chapter, we discuss about the work done in all previous chapters. It is the chapter where all conclusions and future work are.

# CHAPTER 2

# State of the art

This chapter is divided into two subsections. In the first one, we will discuss about basic concepts and common computer graphics current techniques. These are well known concepts and techniques in computer graphics, which are used in our work. In the second section, specific articles will be explained in depth. These articles are the theoretical basis of this Thesis, and we will focus on the parts that will be used in our approach.

## 2.1 Basic concepts

In the following subsections we will describe basic concepts and techniques. Since our work is deeply related with Newton's method and Bézier patches, subsections 2.1.1 and 2.1.2 are especially important.

### 2.1.1 Newton method

The Newton method is an iterative algorithm used to find successively better approximations to the roots of a real valued function. This method is often used to solve real valued functions very fast when exactitude is not mandatory.

Newton's method requires an initial guess of the result. If this initial guess is within a few iterations of the solution, Newton's method might be already close enough to the root. Basically, how quickly and precise are the solutions are depends on the problem. Unfortunately, when iteration begins far from the desired root, Newton's method can easily become unstable or even not converge in a solution. The biggest advantage in computer science is the use of easy computation, using only a sum, a derivative and a fraction in every step. The main idea of Newton's method is summarized in Fig. 2.1.

Given a function $f(x)$ and its derivative $f'(x)$, we begin with a first guess $x_0$.

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{2.1}$$

The process is repeated until a sufficiently accurate value is reached :

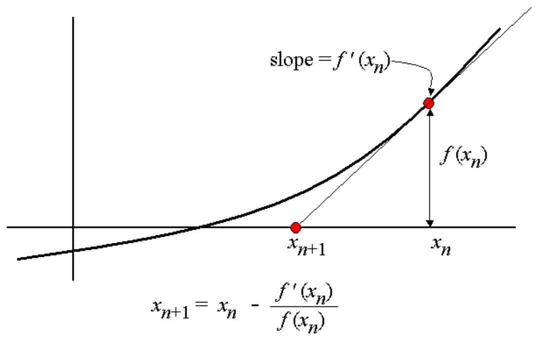


Figure 2.1: Generic iteration of Newton's method.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.2}$$

There are essentially two practical considerations that have to be taken into account when using this method:

- The convergence is not assured. More specifically, when the derivative is zero or close to zero, the tangent line is nearly horizontal, so the solution found will then be even further than in the previous iteration.
- If the initial value is too far from the root, Newton's method may take too many iterations to converge or even fail. For this reason, Newton's method is often used as a local technique for small domains of the function. To avoid failure of convergence, most implementations put a limit on the number of iterations.

Those considerations are solvable and we will talk about them on further sections.

### 2.1.2 Bézier surfaces

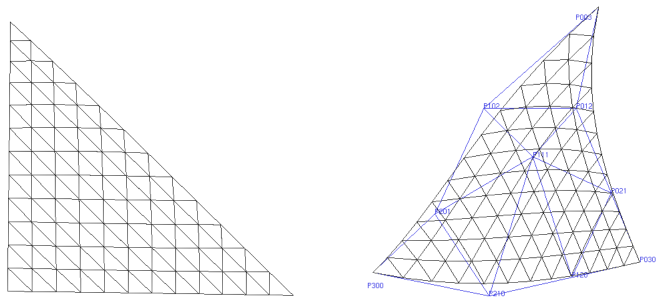
A Bézier surface is a mathematical description of a surface very used in computer graphics. It has many variants developed during the years, that restrict the surface behavior or the data input.

Normally, Bézier surfaces used in 3D computer graphics have a function $B(u, v)$ that transforms coordinates in a 2D space into a 3D space combining parameters in 3D. This 2D space is known as *Parametric Space*, and the 3D parameters are also called *control points.* The number of control points and the way the $B$ function is built determines the properties of the surface, but what they all have in common is that the resultant surface adapts its shape to the given control points, as can be seen in Fig. 2.4, where the blue mesh in Fig. 2.2(b) contains the control points of the Bézier surface.

The general B(u,v) function can be described as:

$$B(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} F_i^n(u) F_j^m(v) k_{i,j} \tag{2.3}$$

where $k_{i,j}$ are the parameters or control points and $F$ is:



(a) Mesh in parametric coordinates.

(b) Resultant 3D mesh.

Figure 2.2: Coordinates transformation.

$$F_i^n(x) = \binom{n}{i} x^i (1-x)^{n-i} \tag{2.4}$$

Bézier surfaces can be of any degree but bicubic Bézier surfaces generally provide enough degrees of freedom for most applications. In Fig. 2.3 we see an example of a bicubic Bézier surface.

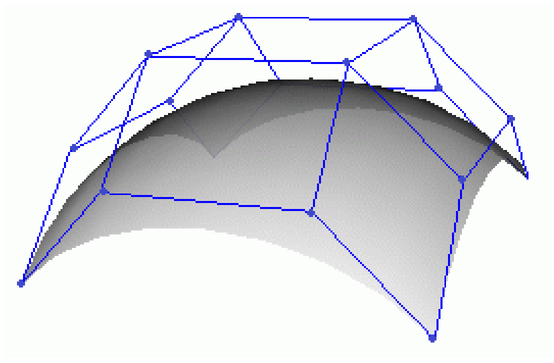
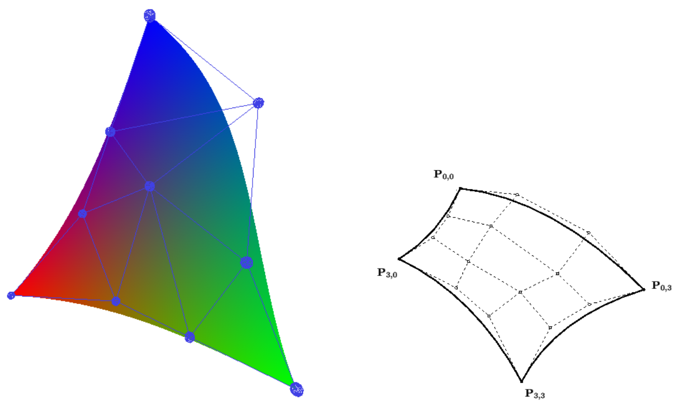


Figure 2.3: Bicubic Bézier surface.

When we talk about a Bézier patch, we mean the portion of the surface where the parametric coordinates have values from 0 to 1, both included. This way we can distinguish a triangular Bézier patch as in Fig. 2.4(a) from a rectangular patch as in Fig. 2.4(b), for example.

The difference between a rectangular and a triangular patch comes from the $B$ function and the number of control points. A rectangular patch can be expressed using two triangular patches, and a triangular patch can be expressed with a trimmed rectangular patch. Trimming a Bézier patch means to restrict the parametric coordinates that can be used.

In computer graphics, Bézier patches are still used, as they are much smoother and compact than triangle meshes. Unfortunately, their difficulty to rasterize or calculate intersections with them make rendering them complicated.



(a) Triangular Bézier patch. (b) Rectangular Bézier patch.

Figure 2.4: Bézier patches.

### 2.1.3 Bump mapping

Bump mapping is a computer graphics technique to make a rendered surface look more realistic rendering less geometry. The main idea is to render the details of a high resolution mesh into a texture and then apply it to a low resolution version of the mesh combined with an algorithm executed in the GPU. Bump mapping is usually combined with other techniques, such as texture mapping. The combination of these two gives the application an excellent visual quality with much fewer triangles, which leads to lower memory and bus bandwidth use and, finally, a better performance in rendering speed.

Bump mapping is the common name for three different techniques: normal mapping, displacement mapping and parallax mapping, explained in sections 2.1.3.1, 2.1.3.2 and 2.1.3.3 respectively. The difference between the methods is the algorithm executed in the GPU and the place in the graphics pipeline.

#### 2.1.3.1 Normal mapping

Normal mapping is a bump mapping techinque used for faking the lighting of bumps and dents. This technique is used so that much more detail can be seen without using more polygons.

The idea is very simple. From a high resolution mesh we generate a texture that codifies the normal orientation in each fragment. Then we apply this texture to a low resolution version of the model and, using a fragment program, we substitute the normal obtained by low resolution geometry for the one codified in the texture before lighting calculations. This way, all low resolution fragments have the same normals as the high resolution ones, and thus, the shading reveals many details lost in the simplification of the mesh. In Fig. 2.6 we can see a comparison between a high detailed mesh, a low detailed mesh without normal mapping and the same low detail mesh rendered using normal mapping.

There are two variants to generate and apply the normal texture: in object space and in tangent space. We describe them now.

**2.1.3.1.1 Object space** Object space is a reference system to represent normals. This reference system is aligned with the whole object, so all normals in the object are expressed from the same coordinates system.

This normal space has advantages for shading:

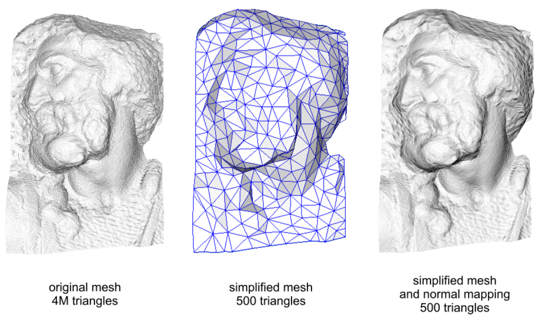


Figure 2.5: Example of normal mapping.

- No need to transform coordinates for light shading.
- Easy implementation for static objects.

Nevertheless, it has its disadvantages:

- When using animated objects, normals have to be stored for every key-frame.
- Higher time overhead and lower precision for animation due to interpolation between key-frames.
- Hard to reuse normals with other differently shaped objects.

In object space, normals don't have to be transformed. It's the simplest way to implement normal mapping. The biggest inconvenient is animation in objects. If an object is animated, a normal texture has to be stored for every key-frame and then interpolated. Clearly it's not an efficient method since a lot of memory will be needed and there will be an overhead for interpolation. For this reasons, object space is more commonly used in inanimated objects.

**2.1.3.1.2 Tangent space** Tangent space is the other reference system to represent normals. This reference system is aligned with the plane tangent with the surface on every point.

As lighting is usually expressed in world space, transformations have to be made to have coordinates space coherence. Converting light rays to tangent space is most common.

This normal space has several advantages for shading:

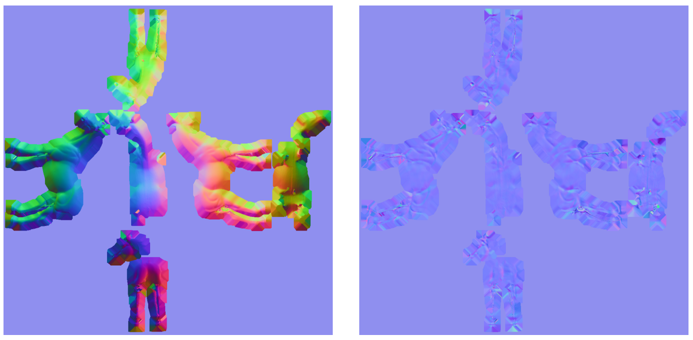
- No need to adapt any algorithm to support animation.
- Higher normal data compression rates.
- Maps created in this space can be reused easily in other shapes.

And it also has its disadvantages:

- Slightly lower performance than object space.
- Harder to avoid smoothing problems and harder to interpolate correctly.

**2.1.3.1.3 Comparison between object space and tangent space** For animated ones, tangent space is the best option, though not exempt of problems. Light rays are in world space, but the normals stored in the normal map are in tangent space. When a normal-mapped model is being rendered, the light rays must be converted from world space into tangent space, using the tangent basis to get there. At that point the incoming light rays are compared against the directions of the normals in the normal map, and this determines how much each pixel of the mesh is going to be lit. Alternatively, instead of converting the light rays some shaders will convert the normals in the normal map from tangent space into world space. Then those world-space normals are compared against the light rays, and the model is lit appropriately.

Figure 2.6 shows the difference



(a) Normal map in object space. (b) Normal map in tangent space.

Figure 2.6: Normal spaces comparison.

### 2.1.3.2 Displacement mapping

Displacement mapping is a bump mapping technique that uses a texture or height map to displace vertices in a mesh to change its shape. One of the first architectures including displacement mapping is described in the article [4]. It's mostly used to create details in low resolution meshes to make it as similar as possible as the high resolution version of the mesh. It is very useful to represent and compress terrain information, as in Fig. 2.7. In this case, the texture is called height map, and the vertices in the mesh are displaced vertically depending on the grayscale value in the height map.

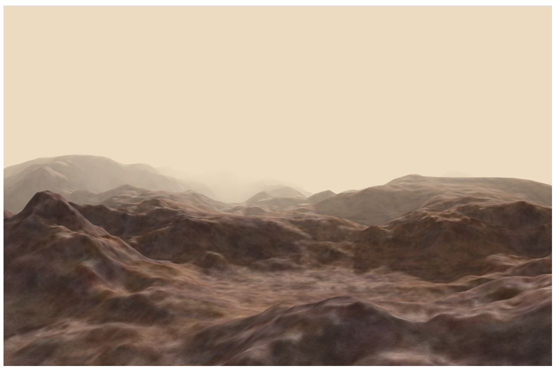
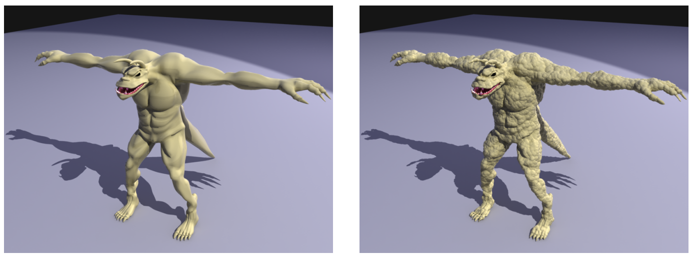


Figure 2.7: Example of height map terrain.

Displacement mapping is also used to create details in objects. In this case, they are combined with a progressive view dependent tessellation of the mesh sent to render. The new vertices created by the tessellation are displaced following an algorithm that uses the texture that contains the details of the high resolution mesh to make the tessellated mesh look like it. In Fig. 2.8 we see the comparison between using displacement mapping or not in an object.



(a) Without displacement mapping. (b) With displacement mapping.

Figure 2.8: Comparison between using or not displacement mapping.

This technique can be performed either in CPU or in GPU, but CPU versions are too slow, so not very used. GPU versions are comparatively fast, but still too slow for many applications, due to the large amount of time required in the tessellation process and displacement of all vertices when high detail tessellation is performed.

Until recently, displacement mapping in real time was unfeasible due to the complexity and the lack of supporting hardware. Since Nvidia 8800 series, though, geometry shading is available. This hardware allows to program geometry shaders that can create multiple primitives for every primitive sent to the GPU. Using this feature, algorithms can be written to perform displacement mapping by reading the texture and creating multiple primitives and displacing the vertices to create a high detailed mesh in rendering time. This detail can even achieve subpixel precision.

Depending on the application, such level of detail can be necessary. In animation films industry, where real time rendering is not mandatory but visual quality is, this technique is now commonly applied. In gaming industry, though, it is still not very used due to the overhead in rendering time. In any case, real time displacement mapping performs view dependent tessellation, subdividing only the closest primitives to the camera.

Even though it is relatively new and the hardware still is too slow for real time applications, displacement mapping in real time is becoming possible, and some game engines, such as Unigine [1], already support it, as shown in Fig. 2.9.

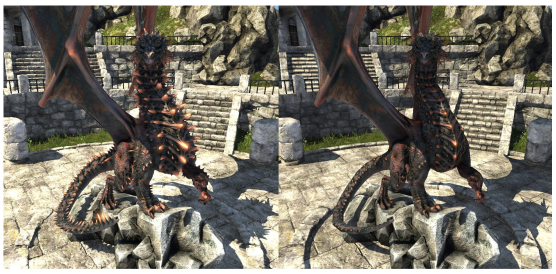


Figure 2.9: Displacement mapping comparison. Image extracted from Unigine engine.

Displacement mapping is supported since shader model 4.0 in Direct3D 10 and OpenGL 3.2, and geometry shaders can be written in the most common shading languages, namely Cg, HLSL and GLSL, among others.

#### 2.1.3.3 Parallax mapping

Parallax mapping is an evolution of normal mapping techniques widely used in many 3D rendering applications and computer graphics in general. The goal is to give more apparent depth and thus, greater realism, to the rendered scene with the same geometry and a low impact on performance. It was first introduced by Tomomichi Kaneko et al. [6] in 2001.

The main idea in parallax mapping is displacing the texture coordinates at a point on the rendered polygon by a function of the view angle in tangent space and the value of the height map at that point. The optical illusion is more visible at steeper view-angles, when the texture coordinates are displaced more, giving the illusion of depth. In Fig. 2.10 we see a comparison of a scene using or not parallax mapping. In the one where parallax mapping is activated, the depth optical illusion indicates that they are very detailed walls, but in fact they are flat polygons.

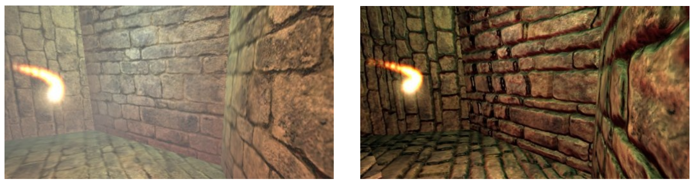

(a) Parallax mapping not activated. (b) Parallax mapping activated.

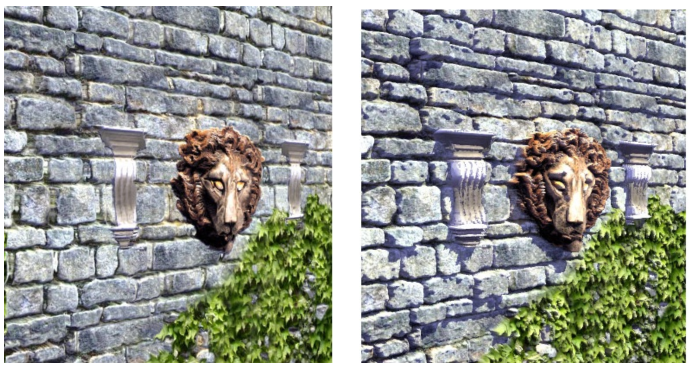
Figure 2.10: Comparison of a scene with and without parallax mapping. Image extracted from Irrlicht 3D engine.

The first version of parallax mapping is a single step process that does not account for occlusion. Further enhancements have been made to incorporate this effects and improve silhouette rendering, in iterative approaches. These enhancements lead to Steep Parallax Mapping technique, by Morgan McGuire et al. in [7]. Steep parallax mapping traces a ray along the height map stored in the texture and iterates over it to find a collision. This means that it is a slightly slower technique, but its results have a better visual quality. In Fig. 2.11 we see a comparison of a scene rendered using parallax mapping and steep parallax mapping. An excellent optical illusion is achieved, since it appears that a complex scene is rendered, but in fact just a single quadrilateral and two textures are sent to the GPU.

#### 2.1.3.4 Comparatives

There are two main criteria to compare the techniques explained before: performance and visual quality. Developers have to choose between them depending on the application and its needs, and unfortunately, the ones with best visual quality are also the slowest. This is the ranking from fastest to slowest and also from lower to better visual quality:

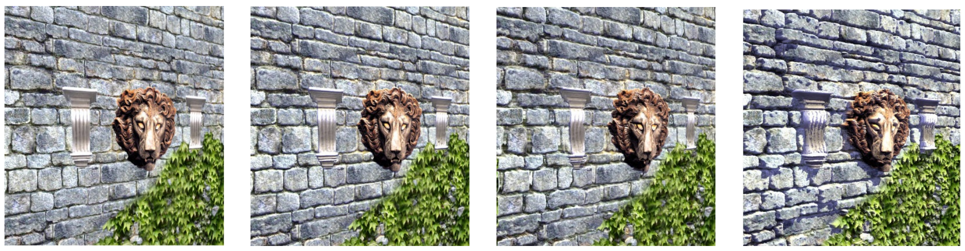
1. Normal mapping.
2. Parallax mapping.
3. Displacement mapping.



(a) Parallax mapped. (b) Steep parallax mapped.

Figure 2.11: Comparison between parallax mapping and steep parallax mapping. Image extracted from Brown University Graphics Group.

Normal mapping and parallax mapping have both very performances and very little overhead time, but parallax mapping delivers a better visual quality, specially steep parallax mapping version. For this reason, parallax mapping is one of the most used techniques is real time rendering. In Fig. 2.12 we see a visual comparison that indicates that steep parallax mapping has a better visual quality.



(a) Texture mapped. (b) Normal mapped. (c) Parallax mapped. (d) Steep parallax mapped.

Figure 2.12: Comparison between techniques.

Nowadays, displacement mapping is rarely used for real time applications because it is a newer technique and geometric shaders are much more recent. Other applications that don't work in real time do take advantage of this technique. In particular, film industry makes an intensive use of it. Its visual quality is much better, and it becomes clear when looking at the silhouettes, as can be seen in Fig. 2.13.

### 2.1.4 Surfaces in modelling

Nowadays many components like car bodies and technical surfaces such as an F1 chassis are designed using Computer Aided Design (CAD) systems. Those programs implement tools for surface designing based on well known mathematical descriptions. Those surfaces are typically Non-uniform rational Basis-splines (NURBS).

When modeling, designers need to have control of two basic aspects: the general shape of the surface, and its curvature. The shape is controlled by various coordinates called Control Points, or even with a polygon called Control Polygon.

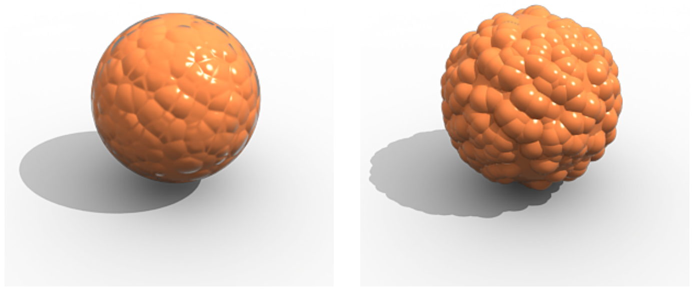

(a) Using normal mapping. (b) Using displacement mapping.

Figure 2.13: Comparison between using or not displacement mapping.

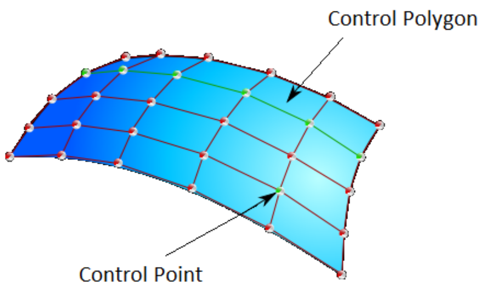


Figure 2.14: Surface design controls.

The normal continuity and smoothness is assured by the mathematical description of the NURBS, in fact, that's the most important reason why this is the most used surface in designing. In the past, Bézier surfaces were very popular, but their smoothness and continuity problems with neighbor surfaces demanded evolutions, so Splines and B-Splines (Basis-Splines) were developed, and then finally NURBS.

#### 2.1.4.1 B-Splines

As described in section 2.2.1, a Bézier surface is a mathematical description for a surface used in computer graphics. A B-Spline is basically a set of connected Bézier surfaces with parametric continuity, which means that no sharp edges connect two Bézier surfaces.

B-Splines came from the big challenge of generating an accurate surface with a single Bézier surface. It was easier to design by segments or piece-wisely than a whole, but then the problem was correctly connecting all parts. B-Splines solve that problem by restricting the coordinates of the control points of all adjacent surfaces. This way, linear continuity is assured.

#### 2.1.4.2 NURBS

Non-uniform rational Basis-Splines (NURBS) is a mathematical model developed to generate and represent surfaces. It is commonly used in computer design and computer graphics because

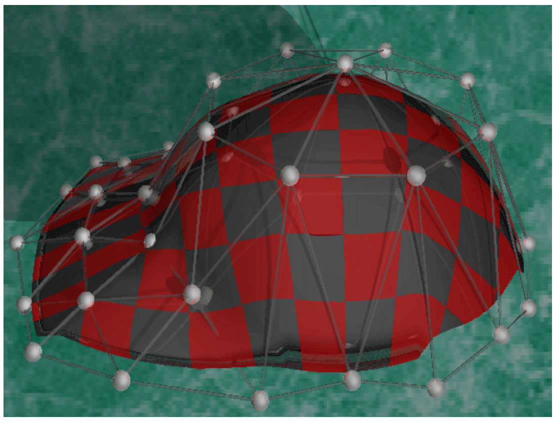


Figure 2.15: Surface modelled using B-splines.

of its great flexibility and precision for handling both analytic and freeform shapes.

As a matter of fact, this need for handling freeform surfaces is the origin of the NURBS, which came as an evolution of B-Splines in the automobile and aerospace industry. NURBS provide more flexibility and precision for surface developing than B-Splines. For example, a sphere can be represented with NURBS, but not with B-Splines.

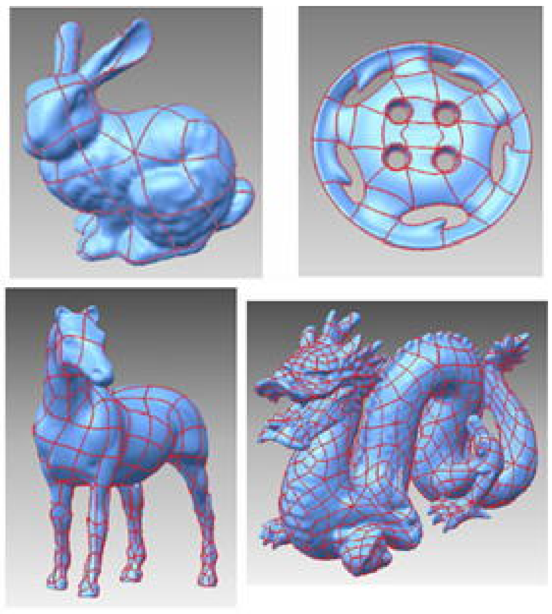


Figure 2.16: Objects modelled using NURBS.

#### 2.1.4.3 Uses and rendering

The most benefited by Bézier surfaces and NURBS is the industry. Engineers can now precisely design every piece from a wing to a screw and even control physical models scanning and comparing them with the virtual ones.

So the use is clearly surface designing, and the rendering is made with basically two different techniques: Ray-Tracing and surface triangle meshing. Ray-Tracing consists in calculating the intersection with the surface of the ray created from the observer to every pixel in the screen, while surface triangle meshing consists in creating a triangle mesh from the surface. This last method is very easy when creating a mesh in the parametric space and then transforming it to 3D space using the $B$ function, as explained in section 2.2.1. Triangle meshing is useful for real-time applications, but for visual quality and correctness, Ray-Tracing is usually better. In Fig. 2.17 we see an example of triangle meshing a square Bézier patch.

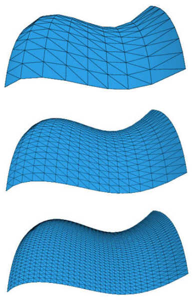


Figure 2.17: Example of triangle meshing.

Some non real time applications developers prefer to use Triangle meshing and subdivide triangles to pixel or even subpixel level to achieve the same precision as Ray-Tracing. This is particularly true for animation films industry. The reason for that is to be able to reuse previous tools and methods developed for triangle meshes.

In Fig. 2.18 we see an example of an object modeled with NURBS using Blender.

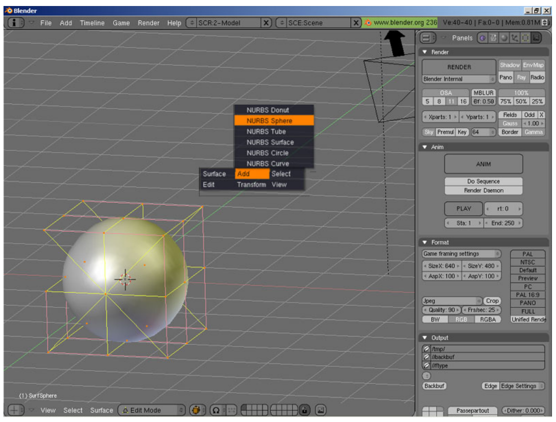


Figure 2.18: Sphere modelled with NURBS. Source Blender.

## 2.2 Previous work

In this subsection we will describe various papers that conform the basis of our work. PN Triangles and its evolution, curved PN Triangles, are the surfaces we use in our approach, and the Newton's method adaptation described in the last two articles is also key in this Thesis.

### 2.2.1 PN Triangles

PN Triangles are basically Bézier triangular patches with a $B$ function that has 10 3D coordinates as parameters, being:

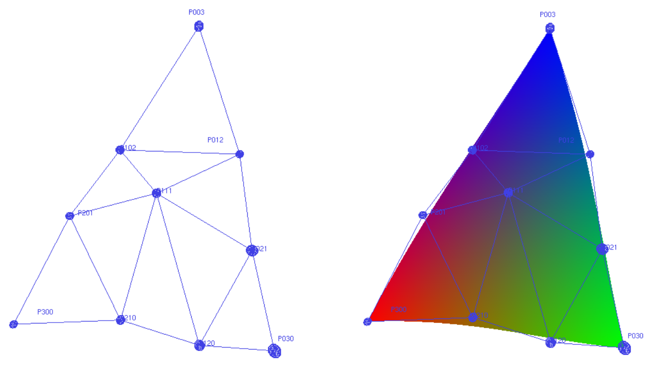
$$
\begin{aligned}
B(u,v) \quad = \quad & b_{300}u^3 + b_{030}v^3 + b_{003}w^3 + \\
& 3b_{210}u^2v + 3b_{120}uv^2 + 3b_{201}u^2w + \\
& 3b_{102}uw^2 + 3b_{021}v^2w + 3b_{012}uw^2 + \\
& 6b_{111}uwv
\end{aligned}
$$

where $w = 1 - u - v$ and where $b_{300}$, $b_{030}$, $b_{003}$, $b_{210}$, $b_{120}$, $b_{201}$, $b_{021}$, $b_{102}$, $b_{012}$ and $b_{111}$ are the 10 points that define the function $B$.

So this function generates Bézier surfaces, but with some interesting properties:

- All surfaces generated include points $b_{300}$, $b_{030}$, $b_{003}$.
- All generated points are included in the convex hull.
- If we consider only values for $u$,$v$ and $w$ between 0 and 1, the limits of the surface will be characterized by $b_{300}$, $b_{030}$, $b_{003}$.

Those properties and the fact that the generated patches are triangular, as shown in Fig. 2.19(b), make them ideal for computer graphics, since nowadays it's deeply related with triangular meshes. In Fig. 2.19 we see a control points mesh and the PN Triangle generated.

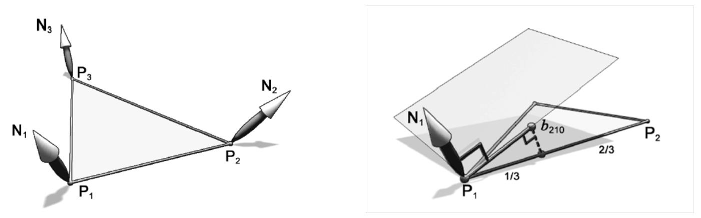


(a) Control points mesh. (b) PN Triangle generated.

Figure 2.19: PN Triangle and its control points mesh.

The contribution of the PN Triangles are the automatic generation of the control points using only local data. The main idea is to place the control points along the plane described by the vertices and normals in the input triangle, as shown in Fig. 2.20

Mathematically, control points are placed following the formulas:



(a) Input triangle. (b) Generated control point coordinates.

Figure 2.20: Automatic coordinates generation.

$$
\begin{aligned}
b_{300} &= P_1 \\
b_{030} &= P_2 \\
b_{003} &= P_3 \\
w_{ij} &= (P_j - P_i) * N_i \in \mathbb{R} \\
b_{210} &= (2P_1 + P_2 - w_{12}N_1)/3 \\
b_{120} &= (2P_2 + P_1 - w_{21}N_2)/3 \\
b_{021} &= (2P_2 + P_3 - w_{23}N_2)/3 \\
b_{012} &= (2P_3 + P_2 - w_{32}N_3)/3 \\
b_{102} &= (2P_3 + P_1 - w_{31}N_3)/3 \\
b_{201} &= (2P_1 + P_3 - w_{13}N_1)/3 \\
E &= (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201})/6 \\
V &= (P_1 + P_2 + P_3) \\
b_{111} &= E + (E - V)/2
\end{aligned}
$$

If the input normals are calculated as the mean of the normals of all triangles surrounding each vertex, some information about the neighbors is stored. This is the information that Gouraud shading technique uses to smooth lighting, and it's the same principle to create Bézier patches that better adjusts to the general shape of the triangle mesh. This way, an object rendered using PN Triangles created from a triangle mesh is reasonably good approximation of how the object would look like if it was modeled using more complex surface models.

### 2.2.2 Curved PN Triangles

Curved PN Triangles [9] is the solution of the biggest problem of PN Triangles, which is the normal discontinuity of neighbor PN Triangles in a mesh, as can be seen in Fig. 2.21.

This happens because PN Triangles build triangular Bézier patches from local information only, not taking into account adjacent surfaces information as we would in NURBS or splines. This way, PN Triangles are built very fast, but the junctures between neighbors are visible because of the normal discontinuity.

The authors propose to apply another triangular Bézier patch to store the normals in the PN Triangle. This patch is quadratic, and not cubic, for simplicity reasons. In Fig. 2.22 we see an example of a quadratic triangular patch.

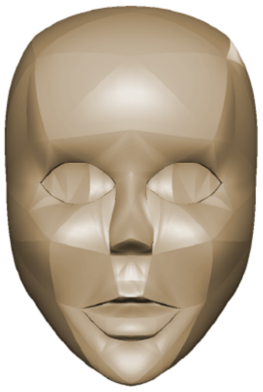


Figure 2.21: Example of normal discontinuities.

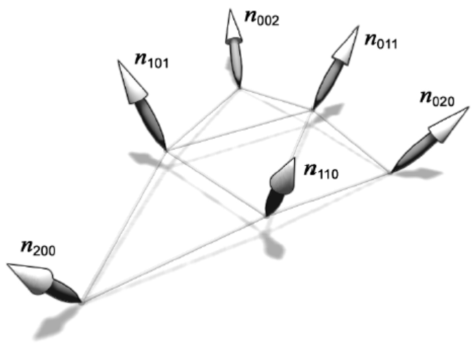


Figure 2.22: Normal triangular patch.

This is how the authors propose to calculate the coordinates of the control points in the quadratic patch:

$$\begin{aligned}
n_{200} &= N_1 \\
n_{020} &= N_2 \\
n_{002} &= N_3 \\
v_{ij} &= 2\frac{(P_j - P_i) * (N_i + N_j)}{(P_j - P_i) * (P_j - P_i)} \in \mathbb{R} \\
n_{110} &= h_{110}/||h_{110}||, h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1) \\
n_{011} &= h_{011}/||h_{011}||, h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2) \\
n_{101} &= h_{101}/||h_{101}||, h_{101} = N_3 + N_1 - v_{31}(P_1 - P_3)
\end{aligned}$$

So, this triangular patch is only used to calculate the normals on every point. The key idea is that all normals on each vertex of the original mesh are created by the sum of all normals of the triangles surrounding the vertex, so this local normal has actually some information of the neighbor triangles. This way, while building the quadratic normal patch for two adjacent triangles $T1$ and $T2$ only with local information, the normals of the vertices in the edge splitting them have the same local information for $T1$ and $T2$, so the normal smoothness is assured.

Briefly, Curved PN Triangles are the combination of two triangular Bézier patches, one for geometry and one for the normals. From the parametric coordinates we can obtain 3D coordinates using the cubic patch and its normal using the quadratic patch. In Fig. 2.23 we see a comparison between a Phong shaded mesh, an object rendered using PN Triangles built from the previous mesh and the same object rendered using curved PN Triangles.

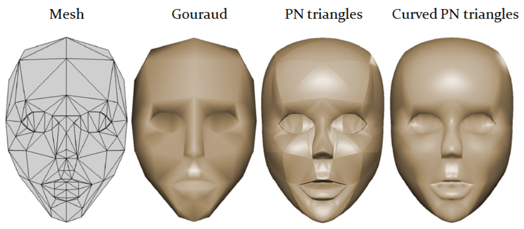


Figure 2.23: Visual results.

### 2.2.3 Generic Mesh Refinement

The article [3] talks about a generic technique to implement a progressive refinement of a mesh. The main idea is to use nowadays graphics capacities to displace vertexes and, using some information about the shape of the detailed surface, approximate the painted mesh.

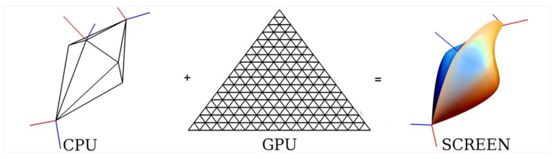


Figure 2.24: Main idea.

In order to do this displacement in a GPU-friendly way, the authors propose to send tesselated triangles as a Vertex Buffer Object (VBO) to the GPU and paint them. Then, design vertex shaders that combine some information to move the tesselated triangles to where they belong. These tesselated triangles are called Refinement Pattern, and the basic simple mesh is called Coarse Mesh.

This information depends on the specific technique used to parametrize the detailed surface. For representing a surface following a sinusoidal pattern, we only need to send the sinus amplitude and the angle for each vertex or write a method in the vertex shader to calculate the angle from some information. The result of this example is Fig. 2.25.

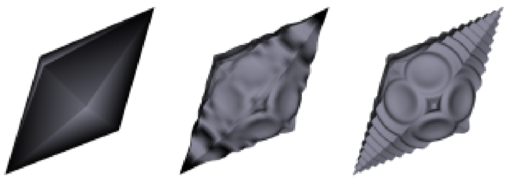


Figure 2.25: Main idea.

If what needs to be refined is a Curved PN Triangle, there are two groups of parameters to send to the shader: the ones that concern the general shape of the Bézier surface, and the ones that concern how each vertex has to be moved. For every triangle of the coarse mesh we calculate its Curved PN Triangle, and in rendering time we send for each coarse triangle its patch descriptions, and for every tesselated triangle of the Refinement Pattern (RP) its parametric coordinates. Then, in the vertex shader we calculate its position thanks to the function that transforms from parametric to 3D coordinates and its normal using the normal patch transform function. In Fig. 2.26 we see a comparison between the coarse mesh and the refined mesh using a 16x16 Refinement Pattern.

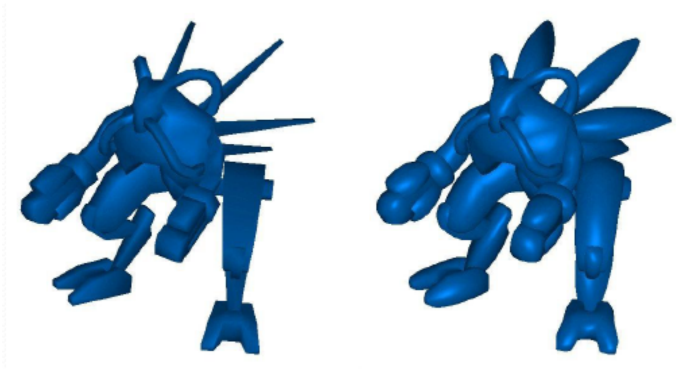

(a) Coarse mesh. (b) Refined mesh using a 16x16 RP.

Figure 2.26: Comparison between Coarse Mesh and refined mesh.

This article proposes a generic method to render progressively refined meshes. The authors state that it is a cheap method to tessellate triangle meshes, but nowadays hardware improves this technique. Its bandwidth use is not negligible, as the authors state, when sending a new RP with all its parameters to the GPU.

### 2.2.4 Interactive Ray Tracing of Trimmed Bicubic Bézier Surfaces without Triangulation

The article [5] presents a technique to render Bézier surfaces interactively using ray tracing. The main idea to render these surfaces quickly is to use Newton's method to find intersections between rays and the surface.

Ray casting is generally a slow technique. One of the first optimizations that are mandatory for real time ray casting is a bounding volume hierarchy. This allows to discard intersections very quickly performing a simple and fast ray-volume intersection. In their case, they use AABB (Axis Aligned Bounding Boxes) because it adapts well into the other parts of their algorithm. This is done in a preprocess.

Once a ray has tested positive for ray-bounding volume intersection, the actual ray-Bézier surface intersection must be computed. This consists in solving the system of equations:

$$\begin{cases} B(u,v) = P \\ P \in ray \end{cases} \tag{2.5}$$

where $B(u,v)$ is the cubic Bézier function, and $u$ and $v$ the parametric coordinates, our unknowns.

Solving this system has two major problems: $B(u,v)$ is a cubic function, which means expansive calculations, and finding a proper representation for the ray.

The authors propose to represent the ray as the intersection of two orthogonal planes:

$$\begin{cases} N_1 P + d_1 = 0 \\ N_2 P + d_2 = 0 \end{cases} \tag{2.6}$$

where $N_1$ and $N_2$ are the normals of the planes and their dot product is 0. $P$ is a point in the ray.

Now, they can express the system of equations the following way:

$$\begin{cases} N_1 B(u,v) + d_1 = 0 \\ N_2 B(u,v) + d_2 = 0 \end{cases} \tag{2.7}$$

At this point, solving this system of equations is still too expansive because of the cubic function. Now, what the authors propose is to use Newton's method in order to accelerate the computation. This method requires a small adaptation in order to apply it in the problem.

The first element needed to adapt Newton's method is to define the function to solve. In this case, the function is the system of equations that needs to be solved. The authors build a matrix and call it R:

$$R = \begin{pmatrix} N_1 B(u,v) + d_1 \\ N_2 B(u,v) + d_2 \end{pmatrix} \tag{2.8}$$

The next element is the Jacobian matrix, which has the partial derivatives of both variables for both equations:

$$J = \begin{pmatrix} N_1 Bu(u,v) & N_1 Bv(u,v) \\ N_2 Bu(u,v) & N_2 Bv(u,v) \end{pmatrix} \tag{2.9}$$

where $B_u$ and $B_v$ are the partial derivatives in the corresponding parametric direction.

With all these elements together, the Newton iteration can be expressed as:

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - J^{-1} R(u_n, v_n) \tag{2.10}$$

Now, the initial guess must be close enough in order to find proper solutions. Provided that a bounding volume hierarchy has been precomputed, the authors propose to use the center of the enclosed parametric domain as the initial guess for the Newton iteration.

At this point, the authors have a fast method to perform a fast ray tracing of Bézier surfaces. But rendering Bézier surfaces themselves might not be very useful. The goal is to render objects described with Bézier surfaces.

The authors render objects described with bicubic square Bézier patches. One of the problems that appear using square patches is that there are unfeasible shapes (i.e. a triangle). Therefore, they need a method to trim the patches.

During the preprocessing step, surface areas are classified and marked if trimming is needed. The area to be trimmed is also calculated in preprocessing time and defined with a curve in parametric coordinates. Then, in rendering time, for those areas that do need to be trimmed, an additional test is performed. This test is a 2D point-in-curve, positive when the point is inside the curve. This way, only points within the trimmed parametric coordinates will be rendered. The result is that all shapes can be rendered.

This ray casting is performed in CPU, and thus, it is still slow. Nowadays processors are multi cored, which means that they are capable of parallel task performing. The authors take advantage of the parallelism using SIMD (Single Instruction Multiple Data) instructions. These kind of instructions mean basically that every instruction is executed independently from the rest, but the whole system shares the data. This suits perfectly for ray casting, since every ray can be processed independently from the rest, sharing only the scene data. Using a cluster of standard PC and this technique, Ingo Wald et al. in [11] and [10] achieve a more than an order of magnitude speed up.

The results shown in Tab. 2.1 show how this technique achieves interactive rendering times using very little memory consumption for even complex models. Models rendered are shown in Fig. 2.27.

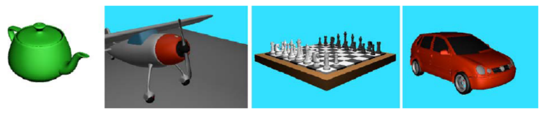


Figure 2.27: Models rendered for results.

| **Models** | **Patches** | **Trims** | **B-Boxes** | **Memory consumption** | **Preprocessing time(min)** | **Average Framerate** |
|---|---|---|---|---|---|---|
| Teapot | 32 | - | 2736 | 150 kB | 0:01 | 9.1 fps |
| Cessna | 1555 | - | 53216 | 3.2 MB | 0:03 | 4.2 fps |
| Chezzboard | 16182 | - | 227794 | 15.6 MB | 0:13 | 6.7 fps |
| VW Polo | 11576 | 38556 | 448500 | 28.3 MB | 5:26 | 5.1 fps |

Table 2.1: Results table

Although it is a very fast technique, it still has drawbacks. First, using quadrilateral Bézier patches limit the shapes that can be modeled, and trimming them adds an overhead in rendering time. Furthermore, the authors do not use graphics hardware acceleration, as they execute all the algorithms in CPU. Also, these framerates are achieved rendering at a fixed resolution of 512x512 pixels, which is very low for nowadays standards.

### 2.2.5 Practical Ray Tracing of Trimmed NURBS Surfaces

The article [2] presents a technique to render NURBS interactively using ray tracing. It is similar to the work described in section 2.2.4. Like in previously described article, the authors take advantage of Newton's method to find intersections between rays and the surface very fast.

The main differences between both articles are the surfaces rendered and the time to render them. In previous case are trimmed square Bézier patches, and in this one the authors propose to render trimmed NURBS. Also, this technique is not performed in real time, as is the previous one.

The article is divided in three parts:

1. Ray tracing NURBS

2. Trimming

3. Results

Ray tracing process is very similar to the previous article. At preprocessing time, NURBS are flattened. The idea is very similar to the Bézier patch subdivision, and it is done to assure that Newton's method will converge quickly. To flatten the NURBS, the mesh control is refined. When refining it, the authors subdivide the NURBS patch into several subpatches. This allows to create a VBH (bounding volume hierarchy) used to store the initial solutions for Newton's method. The subdividing and refining of the control mesh is done until each subpatch meets some flatness criteria, so the authors can assure it is a good initial guess.

These criteria is based in heuristics that work with the curvature of the knot vectors. This heuristic value is given by:

$$n_1 = C_1 * max_{[t_1,t_1+1)}[curvature(c(t))] * arclen(c(t))_{[t_1,t_1+1)} \tag{2.11}$$

which is basically a measure of the B-spline $c(t)$ curvature.

The authors also use AABB as bounding volumes and the ray-NURBS intersection is calculated using the same adapted Newton's method as in the previously described article.

The second main part of the article is trimming. To perform the trimming, the authors use curves that consist in piecewise linear segments in parametric space. The orientation of the curves determine which region of the surface is to be kept. In order to be able to any shape, they build trimming hierarchy. In Fig. 2.28 we see an example of how a trimming hierarchy is built.

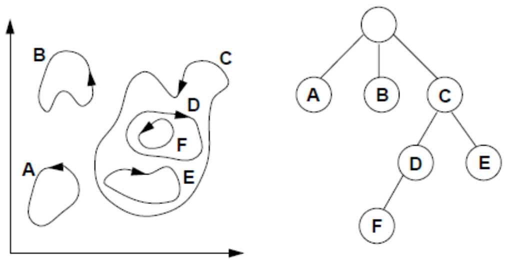
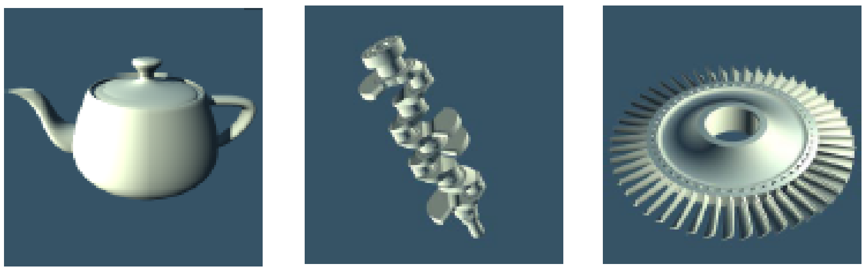


Figure 2.28: A set of trimming curves and the resulting hierarchy. Image extracted from [2]

The third main part of the article is the results. In Tab. 2.2 we see some of the most relevant results of the executions of this technique for the models shown in Fig. 2.29



(a) Teapot. (b) Crank. (c) Allblade.

Figure 2.29: Models used in the results.

| **Statistics** | **Teapot** | **Crank** | **Allblade** |
|---|---|---|---|
| Number of surfaces | 32 | 73 | 351 |
| Number of trims | 0 | 64 | 0 |
| Total time(sec.) | 14 | 40 | 61 |
| Total NURBS time(sec.) | 9.35 | 20.82 | 43.46 |
| Avg time per NURBS(sec.) | 2.17 e-5 | 1.59 e-5 | 1.86 e-5 |

Table 2.2: Results table

In Fig. 2.30 we see a scene that contains NURBS primitives and has been rendered using the authors method.

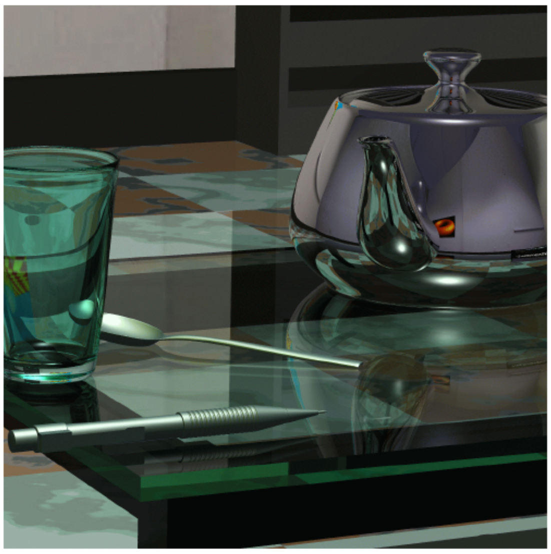


Figure 2.30: Scene ray traced using the authors method. Image extracted from the article [2]

This article has some drawbacks. The most important is that it does not work in real time. It is a very fast method to render NURBS, but still the previous article can work in interactive framerates.

Also, a very important part in the article is related to how to subdivide the NURBS to guarantee the convergence of Newton's method. Although it is true that Newton's method works better subdividing the surface, their heuristic method does not mathematically assure the convergence nor the correctness of the solution found.

CHAPTER 3

# Implementation

In this chapter we will see all necessary implementation details to understand how the Thesis works, including various definitions. This chapter is key for the Thesis results to be understood.

## 3.1 Ray tracing method

Our goal is to render curved PN Triangles in real time. Our options are essentially two: ray casting the Bézier patches or remeshing into a triangular mesh. As explained in section 2.1.4.3, triangle meshing consists in creating a triangle mesh from the surface to render it using conventional algorithms and techniques. If we want to render the triangle meshed patch in real time, only a limited amount of triangles can be obtained due to memory and processing time limitations. Therefore, meshing the patches has some drawbacks:

- Imprecise silhouettes.
- Finite precision when zooming in.

For these reasons, we choose to use the ray casting method. It's not exempt of drawbacks also, namely it's rendering speed. In fact, our challenge is being able to ray cast curved PN Triangles scenes with interactive framerates.

Ray casting a scene fast is a challenge itself, but since PN Triangles have cubic Bézier functions, calculating the intersection of a ray with the Bézier patch is a time-consuming task, so intersecting all rays becomes an even greater one.

The exact calculation of the ray-surface intersection consists in solving the cubic Bézier function for every ray and every frame. This computation is too expensive in time and thus, our first step is simplifying it.

Newton's method offers the possibility of solving ray-surface intersections using only sums, differences and multiplications. In section 3.1.3 we describe how to adapt Newton's method to our problem. This method offers very fast results, but has mainly one problem: it needs sufficiently good initial approximations in order to get correct results.

A good initial approximation for the Newton method is a $u$ and $v$ close to the solution $(u\prime v\prime)$ such that applying the Bézier function $B(u\prime, v\prime)$, the obtained point is in the ray. In order to get good initial approximations, we need an automatic method.

In a triangle, the barycenter is the center of mass, which means that it is placed in the mean of all points coordinates. For our purposes, it's the point that reduces the distance the most to all other points in average, so it's the best initial approximation for our Newton's method implementation. Still, the barycenter could be a bad initial approximation for the furthest points and thus, lead to a bad ray-surface intersection.

The solution to this last problem is to subdivide the Bézier patch into smaller sub-patches and store the barycenter of the sub-patch as the initial approximation for that area of the surface. If the patches are not deformed in rendering time, this can be calculated in preprocessing time. The details about how to subdivide are explained in section 3.1.4. This allows us to have better initial approximations for all points. Note that we are interested in the initial approximations only, so the sub-patches are not stored. Theoretically, there is no limit for how many subdivisions can be created from a Bézier patch, which means that there is no limit for the precision. In chapter 4, we show that there is a practical limit.

Once the calculation problem is out, we can focus on improving ray tracing rendering itself. Checking for every ray the intersections for all Bézier patches or sub-patches is not a very good solution because, even though the use of Newton's method boosts the speed, it is still too slow. The first improvement will be to include bounding volumes and perform a very fast ray-bounding volume test before performing the ray-patch intersection if positive. Bounding volume creation can be done in preprocessing time because a bounding volume does not change in rendering time, so we are not time restricted to create them.

Ray-bounding volume test and the rendering can be performed in a traditional way, in CPU, or in a more modern way, in the GPU using the depth buffer and programming a fragment shader. Following subsections 3.1.1 and 3.1.2 describe the details of both ways to solve the problem.

### 3.1.1 CPU Ray-bounding volume test

In our case, axis aligned bounding boxes (AABB) are a fairly good option because they are very fast to test an intersection with and trivial to build. This fact overcomes the drawback of not spatially adjusting particularly well to the Bézier patch and thus, giving many false positives. Also, this drawback is less determinant the more subdivided the patches are.

Bézier triangular patches have the interesting property that all the points in the patch are contained in the convex hull of the control points, which means that an AABB built to contain them is also containing all Bézier patch points. This way, AABB are valid for our purposes.

Once AABB are found, the next step is defining a fast ray-AABB intersection test. We implement the algorithm described in article [12]. The method consists in first expressing the ray as an origin point $P_o$ plus a displacement along a vector $\lambda v$, decomposed as:

$$\begin{cases} P_x = P_{o_x} + \lambda \vec{v_x} \\ P_y = P_{o_y} + \lambda \vec{v_y} \\ P_z = P_{o_z} + \lambda \vec{v_z} \end{cases} \tag{3.1}$$

The origin point for all rays is the camera position, and $\vec{v}$ is the normalized vector from the origin to the pixel position in scene coordinates.

Once the rays are defined, the method begins by calculating the $\lambda_i$ to all supporting planes of the AABB in every axis $i$. We call minimum planes the ones that have lowest coordinate value for each axis in the AABB, and we call maximum planes the other three. As AABB are aligned with with the axis, $\lambda_i$ can be calculated using:

$$\begin{cases} \lambda_{min_x} = \frac{(AABB_{min_x} - P_{o_x})}{\vec{v_x}} \\ \lambda_{min_y} = \frac{(AABB_{min_y} - P_{o_y})}{\vec{v_y}} \\ \lambda_{min_z} = \frac{(AABB_{min_z} - P_{o_z})}{\vec{v_z}} \end{cases} \tag{3.2}$$

and

$$\begin{cases} \lambda_{max_x} = \frac{(AABB_{max_x} - P_{o_x})}{\vec{v_x}} \\ \lambda_{max_y} = \frac{(AABB_{max_y} - P_{o_y})}{\vec{v_y}} \\ \lambda_{max_z} = \frac{(AABB_{max_z} - P_{o_z})}{\vec{v_z}} \end{cases} \tag{3.3}$$

where $AABB_{min_i}$ and $AABB_{max_i}$ are the minimum maximum values of the AABB vertices in the axis $i$.

Then, we select the maximum value of $\lambda_{min}$ from the three minimum planes and the minimum value of $\lambda_{max}$ from the three maximum planes.

A ray intersects the AABB if and only if $\lambda_{min} < \lambda_{max}$. If the condition is not satisfied, there is no intersection, and if it is, the intersection is located in $P_o + \lambda_{min}\vec{v}$.

### 3.1.2 GPU Ray casting

In GPU ray tracing we don't have to worry with ray-bounding volumes intersection tests algorithms because it is done by hardware with the depth buffer. Every pixel that passes depth test, would pass also a ray-bounding volume test. This way we can improve our bounding volumes to make them more efficient and more adjusted to the Bézier patch. In section 3.1.2.2 we explain the bounding volumes used.

Ray casting itself is performed in a fragment shader. There is some information that the shader needs in order to be able to trace rays in every fragment, which is: the observer 3D coordinates and the fragment 3D coordinates. These coordinates can be calculated in CPU and sent, or calculated directly in GPU. For performance reasons, the best option is the second.

Observers coordinates can be retrieved from camera position and sent to the GPU with a uniform variable, but calculating the fragments 3D position is not so trivial. The technique to obtain them comes from real time volume rendering, and the idea is painting the bounding volumes vertices with specific colors in a way that the fragment position can be interpolated locally using the interpolation of the colors as local coordinates information and sending information about the whereabouts of the bounding volume in the scene, so global 3D coordinates can be obtained. In our approach, we have implemented two techniques using two different bounding volumes: AABB (axis aligned bounding box) and triangular prisms.

#### 3.1.2.1 AABB

AABB are the simplest volumes to build. Our only restriction is to assure that the whole Bézier patch has to be contained in the AABB. Containing all control points of the triangular Bézier patch assures that the restriction is accomplished. Hence, we simply have to create an AABB that contains all control points.

Rendering it correctly for our purposes is not so trivial. We need to give the fragment shader enough information to be able to calculate the fragment position in 3D in a very fast way. A possibility could be to use inverse *Projection* and *Modelview* matrices, but there is a faster solution using the AABB vertex colors.

We can set the vertices with the lowest $X$ with red value 0, the vertices with the lowest $Y$ with green value 0 and the vertices with the lowest $Z$ with blue value 0, and on the contrary, color values set to 1. This way, we obtain a colored AABB as in Fig. 3.1. In that figure, control vertices are rendered as blue spheres, and one of them is displaced along the positive $Z$ axis. 3.1(a) is the frontal view of the AABB, and the displaced control vertex can be seen because it is tangent to the faces of the AABB, and 3.1(b) shows the same AABB rotated where the rest of the control points can be seen as they are tangent to the back face.

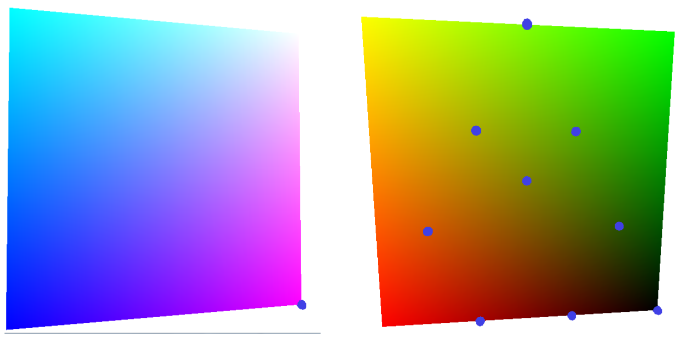

(a) Frontal view of the AABB. (b) Rear view of the AABB.

Figure 3.1: AABB sent to the GPU.

The color interpolation is done automatically by hardware. In the fragment shader we can interpret the color of the fragment as local coordinates. This allows us to place the fragment in object coordinates, but we are still not able to place the fragments in the scene coordinates. The AABB is displaced in the scene by a translation. Countering this translation we could be able to position the fragment in scene coordinates. This can be calculated knowing where the minimum and the maximum vertices are, using the formula:

$$Fragment_{position} = glColor(V_{max} - V_{min}) + V_{min} \tag{3.4}$$

where $glColor$ is the color of the fragment, and $V_{min}$ and $V_{max}$ are the minimum and maximum coordinate values of the vertices in the AABB.

#### 3.1.2.2 Triangular prisms bounding volumes

There is a big overload in fragment processing since that is where intersections of rays with Bézier patches are calculated. Hence, reducing the number of fragments is deeply related with augmenting the rendering speed. Our approach to reduce fragments is to send another bounding volume to render that adjusts better to the Bézier triangular patch. In summary, we have implemented another bounding volume shape to optimize the rendering process.

This shape is a triangular prism. As our Bézier patches are triangular, a triangular prism will adjust better in general than an axis aligned bounding box. This can be false in some cases, but as we comment in other sections, Bézier patches can be subdivided. The more subdivided a Bézier patch is, the more triangular the subpatches are. Therefore, we also have to introduce some criteria to decide if a triangular patch adjusts well to its contained Bézier patch or not.

Our method to build the triangular prism consists in four steps:

1. Finding the equations of the planes for the five faces.

2. Finding the intersections of the planes, which are the vertices of the triangular prism.
3. Creating the structure to be able to send it to the GPU.
4. Coloring the vertices.

The first step begins by finding the orthogonal vector to the vectors $\overrightarrow{P_{300}P_{003}}$ and $\overrightarrow{P_{300}P_{030}}$ using the cross product, as seen in Fig. 3.2. We call this vector $\overrightarrow{N}$.

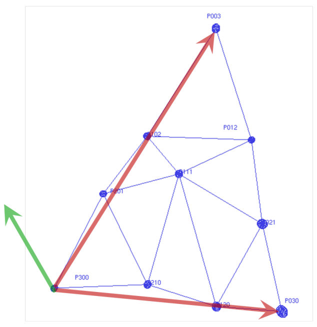
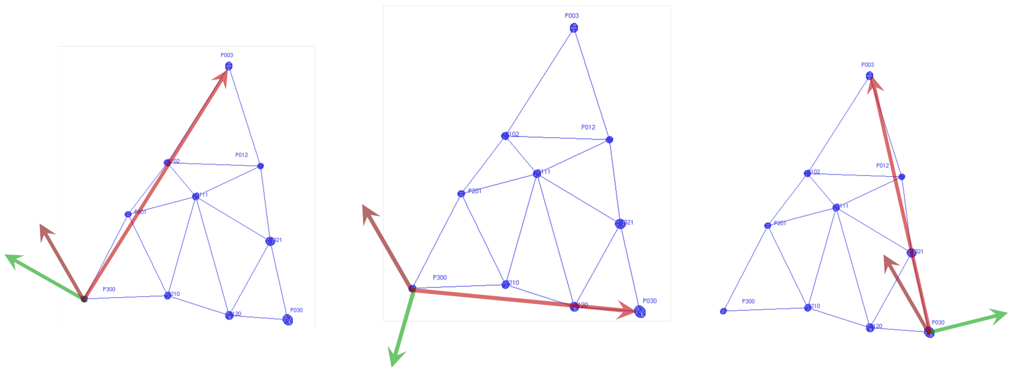


Figure 3.2: Calculating the normal.

We use $\overrightarrow{N}$ as the normal for one of the triangular faces and $-\overrightarrow{N}$ for the other. Once $\overrightarrow{N}$ is found, we can calculate the other three normals. Basically we calculate the cross product of $\overrightarrow{N}$ and $\overrightarrow{P_{300}P_{003}}$, $\overrightarrow{N}$ and $\overrightarrow{P_{300}P_{030}}$ and $\overrightarrow{N}$ and $\overrightarrow{P_{030}P_{003}}$ to obtain $\overrightarrow{N_1}$, $\overrightarrow{N_2}$ and $\overrightarrow{N_3}$ respectively. In Fig. 3.3 we see how we calculate them. The obtained vectors are displayed in green.



(a) $N_1$. (b) $N_2$. (c) $N_3$.

Figure 3.3: Calculating the normals of the planes.

Once the normals are calculated, we need to obtain the independent term of the planes. The restriction to build the bounding volume is that in order to fit all Bézier patch in the volume, it has to be convex and all control points must be inside the volume. This can be done iteratively. For every face of the triangular prism, we calculate a provisional independent term of its supporting plane using one of the control points. Then, we calculate the signed distance of every plane to all the control points. Once we find the furthest point to every plane, we recalculate the plane equation with it.

The second step is to find intersections of the planes. Our implementation consists in using the Jordan-Gauss method to solve this system of equations for every set of three intersecting planes, $P_i$, $P_j$ and $P_k$:

$$\begin{cases} N_iP + d_i = 0 \\ N_jP + d_j = 0 \\ N_kP + d_k = 0 \end{cases} \tag{3.5}$$

where $P$ is the point where the three planes intersect, our unknown.

In this set of three planes, there is always the supporting plane of one of the two triangular faces and two other planes supporting a quadrilateral face. All combinations have to be calculated to find the six vertices of the triangular prism.

The third step requires deciding which structure is better. Our shape is composed by two triangles and three quadrilateral faces. Rendering APIs work best with triangles, and even better with triangle strips. Our shape can be expressed using a single triangle strip, so that is what we will use. In Fig. 3.4 we show an unpacked triangular prism and the order to store the vertices to be able to send them as a triangle strip in rendering time. Unfortunately, there is no way to express it as a triangle strip without repeating at least one vertex. If we choose carefully the vertex to repeat, only a triangle degenerated into a segment will be rendered, creating very few fragments, so the impact is negligible.

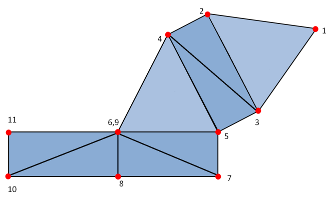


Figure 3.4: Triangle strip of the triangular prism.

The last point is coloring vertices so that the same technique to calculate the fragment position in the scene as we did when using AABB. First, we have to find minimum and maximum coordinate values of the calculated vertices. Then, we assign to each vertex its color interpolating the three components of the coordinates, as in Fig. 3.5:

$$PrismVertex_i.red = (PrismVertex_i.x - min_x)/(max_x - min_x) \tag{3.6}$$
$$PrismVertex_i.green = (PrismVertex_i.y - min_y)/(max_y - min_y) \tag{3.7}$$
$$PrismVertex_i.blue = (PrismVertex_i.z - min_z)/(max_z - min_z) \tag{3.8}$$

### 3.1.3 Newton method applied to PN Triangles

Our method to calculate the intersections of rays with the Bézier patch is very similar to the one described in [5] and commented in section 2.2.4. The goal is to solve the system of equations:

$$\begin{cases} B(u, v) = P \\ P \in ray \end{cases} \tag{3.9}$$

where $B(u, v)$ is the cubic Bézier function of our triangular Bézier patch, and $u$ and $v$ are our unknowns in parametric coordinates.

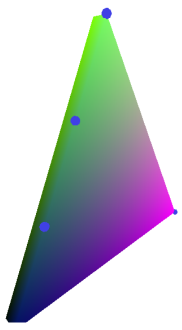


Figure 3.5: Example of a triangular prism.

We can resolve this system using Newton's method expressed as in [5]:

$$\begin{pmatrix} u_{n+1} \\ v_{n+1} \end{pmatrix} = \begin{pmatrix} u_n \\ v_n \end{pmatrix} - J^{-1} \begin{pmatrix} N_1 B(u,v) + d_1 \\ N_2 B(u,v) + d_2 \end{pmatrix} \tag{3.10}$$

where $J$ is the Jacobian matrix containing the partial derivatives of $\begin{pmatrix} N_1 B(u,v) + d_1 \\ N_2 B(u,v) + d_2 \end{pmatrix}$.

In order to accelerate the execution, we can precalculate many coefficients. We can rewrite $B(u,v)$ function to eliminate $w$, since $w = 1 - u - v$:

$$\begin{aligned} B(u,v) \quad = \quad & b_{003}(1 - 3u - 3v + 3u^2 + 3v^2 + 6uv - u^3 - v^3 - 3uv^2 - 3u^2v) + \\ & b_{300}u^3 + b_{030}v^3 + 3b_{210}u^2v + 3b_{120}uv^2 + 3b_{201}(u^2 - u^3 - u^2v) + \\ & 3b_{102}(u + u^3 + uv^2 - 2u^2 - 2uv + 2u^2v) + 3b_{021}(v^2 - uv^2 - v^3) + \\ & 3b_{012}(v + u^2v + v^3 - 2uv - 2v^2 + 2uv^2) + 6b_{111}(uv - u^2v - uv^2) \end{aligned}$$

Now we can reorder it to isolate $u$ and $v$:

$$\begin{aligned} B(u,v) \quad = \quad & (b_{300} - b_{003} - 3b_{201} + 3b_{102})u^3 + \\ & (b_{030} - b_{003} - 3b_{021} + 3b_{012})v^3 + \\ & (3b_{210} - 3b_{003} - 3b_{201} + 6b_{102} + 3b_{012} - 6b_{111})u^2v + \\ & (3b_{120} - 3b_{003} + 3b_{102} - 3b_{021} + 6b_{012} - 6b_{111})uv^2 + \\ & (3b_{003} + 3b_{201} - 6b_{102})u^2 + \\ & (3b_{003} + 3b_{021} - 6b_{012})v^2 + \\ & (6b_{003} - 6b_{102} - 6b_{012} + 6b_{111})uv + \\ & (3b_{102} - 3b_{003})u + \\ & (3b_{012} - 3b_{003})v + \\ & b_{003} \end{aligned}$$

Now, we can rewrite $B(u,v)$:

$$\begin{aligned}B(u,v) = \quad & Q_{30}u^3 + Q_{03}v^3 + Q_{21}u^2v + Q_{12}uv^2 + \\ & Q_{20}u^2 + Q_{02}v^2 + Q_{11}uv + Q_{10}u + Q_{01}v + Q_{00}\end{aligned}$$

where $Q_{ij}$ are constants obtained from grouping $u$ and $v$ factors:

$$\begin{aligned}
Q_{30} &= P_{300} - P_{003} - 3P_{201} + 3P_{102} \\
Q_{03} &= P_{030} - P_{003} - 3P_{021} + 3P_{012} \\
Q_{21} &= 3P_{210} - 3P_{003} - 3P_{201} + 6P_{102} + 3P_{012} - 6P_{111} \\
Q_{12} &= 3P_{120} - 3P_{003} + 3P_{102} - 3P_{021} + 6P_{012} - 6P_{111} \\
Q_{20} &= 3P_{003} + 3P_{201} - 6P_{102} \\
Q_{02} &= 3P_{003} + 3P_{021} - 6P_{012} \\
Q_{11} &= 6P_{003} - 6P_{102} - 6P_{012} + 6P_{111} \\
Q_{10} &= 3P_{102} - 3P_{003} \\
Q_{01} &= 3P_{012} - 3P_{003} \\
Q_{00} &= P_{003}
\end{aligned}$$

And thus, partial derivatives of $B(u,v)$ can be written in the following form:

$$\begin{aligned}
Bu(u,v) &= 3Q_{30}u^2 + 2Q_{21}uv + Q_{12}v^2 + 2Q_{20}u + Q_{11}v + Q_{10} \\
Bv(u,v) &= 3Q_{03}v^2 + Q_{21}u^2 + 2Q_{12}uv + 2Q_{02}v + Q_{11}u + Q_{01}
\end{aligned}$$

#### 3.1.3.1 Solutions tests

As seen in section 2.1.1, the key for this method to work properly is to have good initial approximations, which is why we need to subdivide Bézier patches. This being done, Newton's method can still give wrong results i.e. when a ray does not intersect the Bézier patch. Thus, a test has to be made to check the solution obtained after the iterations.

First, a point inside the Bézier patch can only have coordinates such as $0 <= u <= 1$, $0 <= v <= 1$ and $0 <= w <= 1$. Any obtained solution that is not contained in that range is not valid. Because of limited precision, this condition must be relaxed. In our implementation, we use the condition $-0.0001 <= u <= 1.0001$, $-0.0001 <= v <= 1.0001$ and $-0.0001 <= w <= 1.0001$. As these are parametric coordinates values, it is not dependent on the scale of the rendered scene.

Still, precision errors cause artifacts, so we perform another test. 3D points obtained by performing $B(u,v)$ must be inside both planes that describe the ray. Therefore, the distance from the point to the plane must be 0. We accept small computation errors in this test also, but as this test works in scene coordinates, it is dependent on the scale of the rendered object. The test is the following:

$$\begin{aligned}
abs(dot(N_1, B(u,v)) + d_1) < error \\
abs(dot(N_2, B(u,v)) + d_2) < error
\end{aligned}$$

where $N_1$ and $N_2$ are the normals of the planes that describe the ray, $d_1$ and $d_2$ are their independent terms, and *error* is the scale-dependent accepted error.

### 3.1.4 Bézier patches subdivision

A method to reduce precision errors and other artifacts is subdividing a Bézier patch into multiple sub-patches. The application of this technique will be explained in further sections.

There are infinite ways to subdivide a triangular Bézier patch, even infinite for every number of resultant sub-patches. De Casteljau's method to subdivide patches is well known, and for triangular patches it divides them into three sub-patches. We will first explain the 2D De Casteljau subdivision, which is easier to understand.

For a Bézier curve as shown in yellow in Fig. 3.6 we have four control points $b_1$, $b_2$, $b_3$ and $b_4$, shown in blue. Then, we select a $t$ from 0 to 1, and we take the points $b_i^1$, $t$-proportional of the distance between $b_i$ and $b_{i+1}$, shown in red. We repeat the process using the red points to create the green ones, and we repeat it to create the black one.

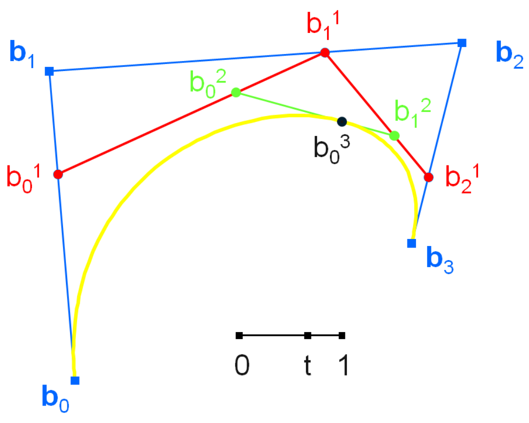


Figure 3.6: De Casteljau 2D subdivision scheme.

By construction, De Casteljau proves that the black point belongs to the Bézier curve, and that the Bézier curve formed with the control points $b_0$, $b_0^1$, $b_0^2$ and $b_0^3$ and the Bézier curve with control points $b_0^3$, $b_1^2$, $b_2^1$ and $b_3$ together form the same curve as the original for any $t$.

For Bézier patches the method can be extrapolated, but with a 2-dimensional $t$. In Fig. 3.7 we show an example of the De Casteljau's method choosing a $t = (0.33, 0.33)$, and in Fig. 3.8 we show the resultant sons.

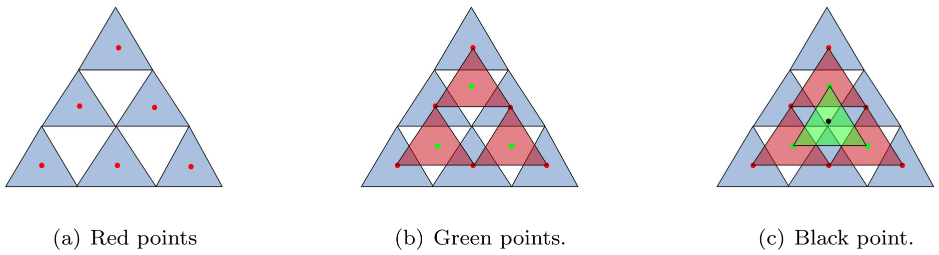
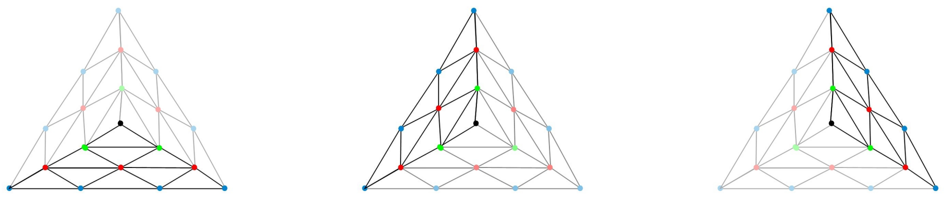

(a) Red points (b) Green points. (c) Black point.

Figure 3.7: Example of a De Casteljau's subdivision.

As seen in the figures, this method creates 3 sub-patches, but the problem is that as they become more and more obtuse as they are recursively subdivided. We chose to divide in half



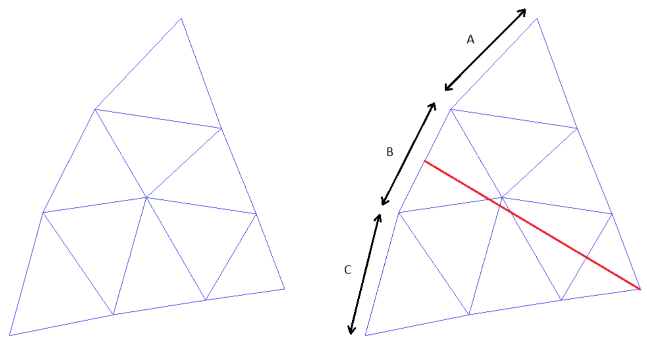
(a) First son (b) Second son. (c) Third son.

Figure 3.8: Example of a De Casteljau's subdivisions resultant sub-patches.

the original Bézier patch into two sub-patches to have a finer control of the total number of patches in a scene. It is also important to prevent having very obtuse sub-patches because it can lead to computation errors due to lack of sufficient precision.

Our approach is to discard one of the sub-patches and construct the remaining two so that they cover all the original patch. This can be done choosing wisely $t$. If we choose $t_1 = (0.5, 0.0)$, $t_2 = (0.0, 0.5)$ or $t_3 = (0.5, 0.5)$, the conditions are satisfied.

Choosing which one of those values for $t_i$ determines the shape of the sub-patches. The best way to avoid having very obtuse sub-patches is to choose to divide the longest side of the Bézier patch in half, as in Fig. 3.9. The length of the sum of all edges of the control points mesh in every side determines which is the longest side in the PN Triangle. In the figure, the longest side is the one formed by the edges of the control points mesh A, B and C.



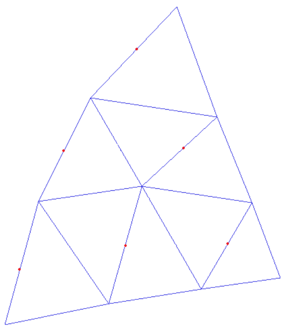
(a) Input control points defining a Bézier patch. (b) Longest edge is the sum of A,B and C. The division is made by the red axis.

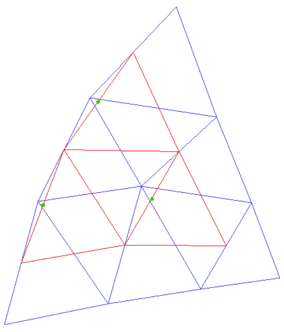
Figure 3.9: Choosing the side to subdivide.

Once the longest side is determined, we can determine $t$, by choosing from the $t_i$ values the one that is in the longest side.

Then, we create the three sets of points: the red points, the green points and the black point. We fill these sets recursively following the De Casteljeau method using the selected $t$, as in Fig. 3.10.

Once all sets are filled, dividing the surface is a question of choosing which ones correspond to each sub-patch, and what we need to create a sub-patch is essentially ten points. Both of the sub-patches will have 4 of the original control points, 3 red points, 2 green points and the black point.


(a) Red points.


(b) Green points.

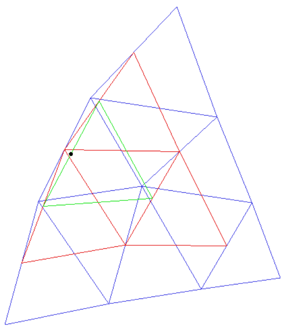
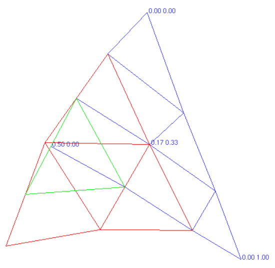
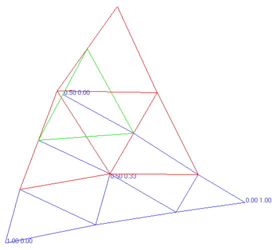

(c) Black point.

Figure 3.10: Set filling process.



(a) First sub-patch.



(b) Second sub-patch.

Figure 3.11: Sub-patches.

The criteria to build these ten-point sets are the following:

- Common for both sub-patches

  Add the black point.

  Add the green point opposite to the green edge where the black point lays.

  Add the red point opposite to the red edge where last added point lays.

  Add the original control point opposite to the control mesh edge where last added point lays.

- For first son

  Add all vertices in the positive subspace in parametric coordinates of the line formed by the last added point and the black point.

- For second son

  Add all vertices in the negative subspace in parametric coordinates of the line formed by the last added point and the black point.

Subdividing Bézier patches is an expensive task in terms of time. Since typically the shape of the PN Triangles do not change in rendering time, it can be done in pre-processing time.

## 3.2 Normal shading

In this section we discuss how to shade the PN Triangles. We propose two methods: one using the approach described in the article [9], and another using normal mapping.

### 3.2.1 Quadratic triangular patch

As discussed in the article [9], PN Triangles don't have normal continuity. For that reason, they propose to use another triangular Bézier patch to calculate the normals in every point. This patch is a quadratic Bézier patch. To distinguish between them, we will call the bicubic Bézier patch as *geometric patch* and the quadratic patch as *normal patch*.

Calculating the normal patch can be done in preprocessing time. As explained in the article, the reflection $A'$ of a vector $A$ across a plane with a normal direction $B$ is $A' = A - 2vB$ where $v = (B \cdot A)/(B \cdot B)$. Using this formula, the authors define the method to calculate the control points using the following formulas:

$$\begin{aligned}
n_{200} &= N_1 \\
n_{020} &= N_2 \\
n_{002} &= N_3 \\
v_{ij} &= 2\frac{(P_j - P_i) * (N_i + N_j)}{(P_j - P_i) * (P_j - P_i)} \in \mathbb{R} \\
n_{110} &= h_{110}/||h_{110}||, h_{110} = N_1 + N_2 - v_{12}(P_2 - P_1) \\
n_{011} &= h_{011}/||h_{011}||, h_{011} = N_2 + N_3 - v_{23}(P_3 - P_2) \\
n_{101} &= h_{101}/||h_{101}||, h_{101} = N_3 + N_1 - v_{31}(P_1 - P_3)
\end{aligned}$$

where $N_1$, $N_2$ and $N_3$ are the normals of the three input vertices.

Then, in rendering time we have to calculate the normal values for each fragment. As discussed in previous sections, using Newton's method we obtain $u$ and $v$. They are used to be able to discard fragments that do not belong to the geometric patch. We can use them also to calculate the normal in each fragment using the $B$ function of the normal patch:

$$B(u, v) = n_{200}u^2 + n_{020}v^2 + n_{002}w^2 + n_{110}uv + n_{011}vw + n_{101}uw \tag{3.11}$$

where $w = 1 - u - v$.

### 3.2.2 Normal mapping

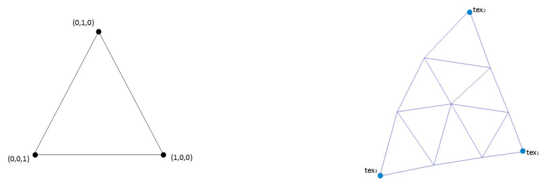
Instead of using the shading proposed in the article [9], we can also shade the PN Triangles using normal mapping. What we need to use this technique, is a high resolution model, a low resolution version of the model, a program to generate the texture map and a shader implementing normal mapping.

In the implementation aspect, the main differences between this technique and the one explained in previous subsection are: reading the model and extracting normals. If we want to use normal mapping, we need to read the texture coordinates for every vertex from the file. Once in rendering time, three pairs of texture coordinates are sent for every PN Triangle, two for every vertex in the original mesh. Then, in the shader the texture coordinates $T$ can be calculated as:

$$T = tex_1 u + tex_2 v + tex_3 w \tag{3.12}$$

where $tex_i$ is a vector of length 2, containing the texture coordinates.

This is the formula to interpolate texture coordinates in a triangle. As we can see in Fig. 3.12, using this formula and using correctly the three read texture coordinates, we can interpolate the texture coordinates.



(a) Correspondance in parametric space. (b) Texture coordinates in 3D space.

Figure 3.12: Texture coordinates interpolation.

Once the texture coordinates are calculated, we extract the texture value, which is a normal value. In our approach, we have implemented normal mapping in object space. This way, the only step left is to calculate the lighting using the normal read from the texture.

## 3.3 Shader optimizations

In this section we will discuss about the optimizations implemented in the shaders. The first optimization increases performance whereas the second one increases visual correctness.

### 3.3.1 MAD

MAD is short for multiply then add. Nowadays graphics cards hardware allow a sum in the same cycle that a multiplication is executed. Therefore, programming shaders that group multiplications and sums is more optimal.

To use MADs we need to decompose the calculations and group them by a multiplication and a sum. In our shader, we have several calculations that can be transformed into MADs, for example:

$$\begin{aligned}
B(u,v) &= Q_{30}u^3 + Q_{03}v^3 + Q_{21}u^2v + Q_{12}uv^2 + \\
&\quad Q_{20}u^2 + Q_{02}v^2 + Q_{11}uv + Q_{10}u + Q_{01}v + Q_{00} \\
Bu(u,v) &= 3Q_{30}u^2 + 2Q_{21}uv + Q_{12}v^2 + 2Q_{20}u + Q_{11}v + Q_{10} \\
Bv(u,v) &= 3Q_{03}v^2 + Q_{21}u^2 + 2Q_{12}uv + 2Q_{02}v + Q_{11}u + Q_{01}
\end{aligned}$$

These are the calculations executed inside every Newton's iteration. Optimizing them would boost the performance, since they are executed several times for every fragment.

Horner's rule states that given any polynomial function:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_0 \tag{3.13}$$

it can be rewritten as:

$$p(x) = ((a_n x + a_{n-1})x + ...)x + a_0 \tag{3.14}$$

This way, our calculations can be grouped the following way:

$$\begin{aligned}
B(u,v) &= ((Q_{30}u + Q_{21}v + Q_{20})u + (Q_{12}v + Q_{11})v + Q_{10})u + \\
&\quad ((Q_{30}v + Q_{02})v + Q_{01})v + Q_{00} \\
Bu(u,v) &= (2(Q_{21}v + Q_{20}) + 3Q_{30}u)u + (Q_{12}v + Q_{11})v + Q_{10} \\
Bv(u,v) &= (2(Q_{12}u + Q_{02}) + 3Q_{03}v)v + (Q_{21}u + Q_{11})u + Q_{01}
\end{aligned}$$

In our shader, the unoptimized code to calculate the $B$ function and its derivatives is the following:

```
dBu=Q30*3.0*u*u + Q21*2.0*u*v + Q12*v*v + Q20*2.0*u + Q11*v + Q10;

dBv=Q03*3.0*v*v + Q21*u*u + Q12*2.0*u*v + Q02*2.0*v + Q11*u + Q01;

Buv=Q30*u*u*u + Q03*v*v*v + Q21*u*u*v + Q12*u*v*v + Q20*u*u + Q02*v*v +
    Q11*u*v + Q10*u + Q01*v + Q00;
```

Code 3.1: $B$ function and its partial derivatives

And rewritten to optimize it using MADs is the following:

```
dBu=Q21*v + Q20;
aux=Q30*u*3.0;
dBu=2.0*dBu + aux;
aux=Q12*v + Q11;
aux=v*aux + Q10;
dBu=dBu*u + aux;

dBv=Q12*u + Q02;
aux=Q03*3.0*v;
dBv=2.0*dBv + aux;
aux=Q21*u + Q11;
aux=aux*u + Q01;
dBv=v*dBv + aux;

Buv=Q21*v + Q20;
Buv=Q30*u + Buv;
aux=Q12*v + Q11;
aux=aux*v + Q10;
Buv=u*Buv + aux;
aux=Q03*v + Q02;
aux=v*aux + Q01;
aux=v*aux + Q00;
Buv=u*Buv + aux;
```

Code 3.2: $B$ function and its partial derivatives written using MADs

### 3.3.2 Multiple seeds

Sometimes Newton's method gives not good enough results. As seen in previous sections, this can happen for several reasons. The result of it is that the fragment shader will detect it wrong and the fragment will be discarded. Therefore, some fragments will not be painted, and thus, the resultant image can have holes.

Avoiding this problem is not trivial. Precision is bounded by hardware, and improving Newton's methods accuracy means executing many more iterations, which leads to lower performance. Nevertheless, there is a way to eliminate the holes without dramatically reducing performance.

As seen in previous sections, placing correctly the initial guess for Newton's method is crucial for the method to converge correctly and in few iterations. Placing it in the barycenter in parametric coordinates of the triangular Bézier patch is the best option in mean for all points, but it might be a poor guess for the furthest. In the vast majority of cases, if the method does not converge fast for a fragment, it means that the fragment is far from the initial guess.

This way, the solution is to have multiple initial guesses (or seeds) for every Bézier triangular patch. Placing these seeds in the furthest points allows Newton's method to converge in many points where it would, otherwise, fail. The furthest points are the vertices of the patch in parametric coordinates.

The difference in the shader implementation is that we try with all seeds before discarding a fragment. If a fragment is discarded after some Newton's iterations, we change the seed and start over. This is done with all seeds, so a fragment is only discarded after failing with all seeds.

Executing this many Newton's iterations leads to a slower performance, but this happens only in a few fragments. In general Newton's method converge with the first seed.

## 3.4 File formats

There are many formats that describe models. In this Thesis we implement a reader that supports *ply* format. It is a well known format in research, and many of the commonly employed models are described in that format.

This format consists in a header, a series of 3D coordinates, and a series of indexes that describe the triangles. If the model has an associated texture for normal mapping, the header is slightly different, and the texture coordinates are described along with the indexes to the vertices. The header is formed in the following way:

```
ply
format ascii 1.0
comments
element vertex n
property float x
property float y
property float z
element face m
property list uchar int vertex_indices
property list uchar float texcoord
end_header
```

where n is the number of vertices and m is the number of faces.

CHAPTER 4

# Results

In this chapter we show the results of various executions of our approach. We divide the results in three steps to be able to compare different aspects. Specially important is the difference in implementing our approach in CPU or in GPU.

All executions are performed using a laptop with a Dual core 2.1GHz processor, 4GB of RAM memory and an ATI Mobility Radeon HD 4330 graphics card. All images rendered have a resolution of 800x600.

## 4.1 CPU ray tracing a single PN Triangle

In this section we will show the results of the CPU implementation executed to render a single triangular Bézier patch. No multi threading or other kinds of optimizations are implemented since the goal is to be able to compare the increment in efficiency by executing the method in GPU. Although optimizing CPU ray tracing can achieve good increments, our approach implemented in GPU can increment performance some orders of magnitude more. There are many reasons for this big increase, but the major reason is that GPU implementation uses a dedicated specific hardware that performs most of the time-consuming operations.

Precision and correctness in our approach depends on various factors, and one of them is the camera position. For example, in Fig. 4.1 we see the same triangular patch rendered from two different points of view. In Fig. 4.1(b) the lower part is rendered incorrectly due to lack of precision.

As performance in time is also view-dependent, in this section all tables show the mean of ten executions from ten different points of view, the same for every execution. Also, to be able to compare properly, all single triangular Bézier patches in this section and the next one have the same control points.

### 4.1.1 Subdivision

To solve precision problems, we can subdivide the patch. This has an impact in rendering. Tab 4.1 and graph 4.2 show how the rendering time depends on the number of subdivisions.

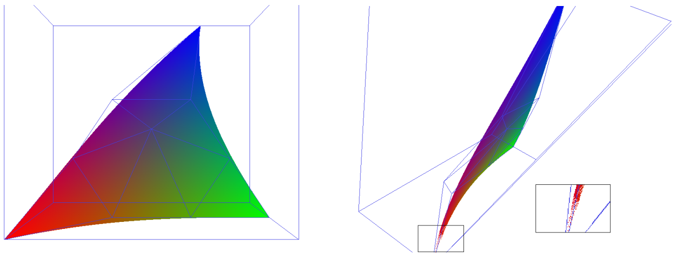

(a) Rendered correctly (b) Rendered with errors

Figure 4.1: Correctness depending on the point of view.

As the subdivisions grow exponentially, the rendering time also does. Ray-box intersections decrease as we increase the subdivision level because bounding volumes adjust better to the patch and more false positives are discarded. Ray-patch intersections decrease slightly because of precision, specially in the silhouette of the patch.

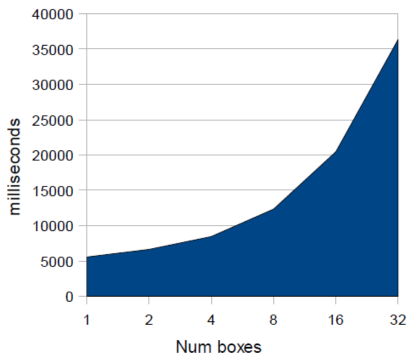


Figure 4.2: Rendering time depending on the number of subdivisions.

| Stats | Num patches | Num sub-patches | Ray-Box intersections | Ray-patch intersections | Rendering time (ms.) |
|---|---|---|---|---|---|
| CPU | 1 | 1 | 262778 | 58030.7 | 4120.7 |
| | 1 | 4 | 189044 | 53361.7 | 4873.9 |
| | 1 | 16 | 171629 | 53054.2 | 10826.9 |

Table 4.1: Comparison between subdivision levels

In Fig. 4.3 we see a close up to an area where artifacts appear. When not subdividing as in subfigure 4.3(a), the artifacts are visible, but subdividing they can be eliminated. Nevertheless, subdividing only is no guarantee of correctness, and it causes a drop in performance. This way, combining subdivision with other methods might be the best option to obtain visual correctness without increasing much rendering time.

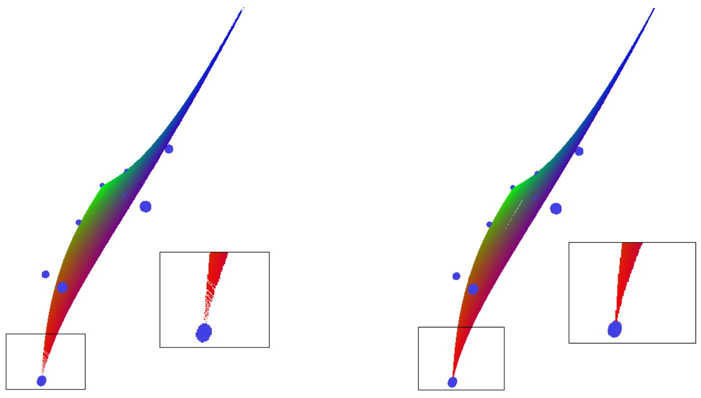

(a) No subdivision (b) 4 subpatches

Figure 4.3: Visual correctness depending on the subdivision level.

### 4.1.2 Performance

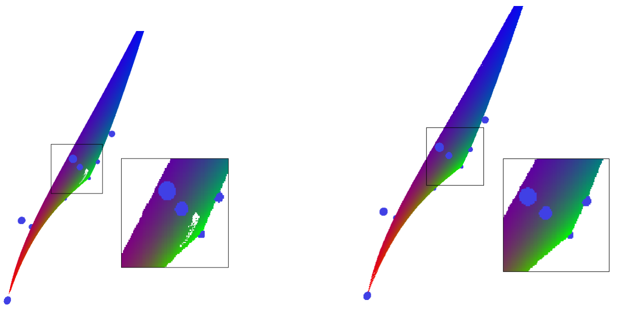
Another method to increase visual quality and reduce artifacts is to increase the number of Newton's iterations. In Tab. 4.2 we show the performance of our approach implemented in CPU comparing with different number of Newton's iterations.

| **Statistics** | **Num patches** | **Num sub-patches** | **Ray-Box in-tersections** | **Ray-patch in-tersections** | **Rendering time (ms.)** |
|---|---|---|---|---|---|
| 4 iterations | 1 | 1 | 262778 | 58030.7 | 4120.7 |
| | 1 | 4 | 189044 | 53361.7 | 4873.9 |
| | 1 | 16 | 171629 | 53054.2 | 10826.9 |
| 6 iterations | 1 | 1 | 262778 | 58172.4 | 6055.5 |
| | 1 | 4 | 189044 | 53938.7 | 6258.6 |
| | 1 | 16 | 171629 | 53036.8 | 12066.3 |
| 8 iterations | 1 | 1 | 262778 | 57722.7 | 7254.5 |
| | 1 | 4 | 189044 | 53671.9 | 7153.7 |
| | 1 | 16 | 171629 | 53011.1 | 13401.4 |

Table 4.2: Executions showing performance in CPU

As was expected, increasing the number of Newton's iterations slows dramatically the rendering speed. In Fig. 4.4 we show the visual difference between executing four Newton iterations and eight. Figure 4.4(a) is a render with four iterations and artifacts can be seen in the lower part. Figure 4.4(b) is a rendering with eight iterations where many artifacts disappear.

These results indicate that to achieve visual correctness, the best solution is a combination of augmenting the subdivision level and the number of Newton's iterations. The problem is that both of the solutions increase the rendering time. From Tab. 4.2 we can obtain the best combination. All non visually correct results have to be discarded. Not subdividing, for example, is not an option for any number of Newton's iterations since the results are not visually correct. For 4 and 6 iterations, both not subdividing and subdividing into 4 subpatches do not give acceptable results. This way, we have as possible combinations: 4 iterations and 16 subpatches, 6 iterations and 16 subpatches, 8 iterations and 4 subpatches, and 8 iterations and 16 subpatches. The fastest option is 8 iterations and 4 subpatches.



(a) 4 iterations (b) 8 iterations

Figure 4.4: Correctness depending on the number of Neton's iterations.

## 4.2 GPU ray tracing a single PN Triangle

Implementing our approach using graphics hardware is more efficient and allows a better performance in rendering time. In this section we display comparisons between CPU and GPU implementations, and between different optimizations of the GPU implementation.

Table 4.3 shows a comparison between the same algorithm, implemented in CPU and GPU. The number of Newton iterations is four for both cases.

| **Statistics** | **Num patches** | **Num sub-patches** | **Ray-Box intersections** | **Ray-patch intersections** | **Rendering time (ms.)** |
|---|---|---|---|---|---|
| CPU | 1 | 1 | 262778 | 58030.7 | 4120.7 |
| | 1 | 4 | 189044 | 53361.7 | 4873.9 |
| | 1 | 16 | 171629 | 53054.2 | 10826.9 |
| GPU | 1 | 1 | 429445 | 95344.7 | 2.5 |
| | 1 | 4 | 256037 | 116375 | 2.7 |
| | 1 | 16 | 193147 | 125090 | 4 |

Table 4.3: Comparison between CPU and GPU implementations

The big difference can be explained mainly by the fact that we use specific hardware to perform operations that are otherwise executed software-wise. For example, in GPU implementation, there is no need to perform a ray-volume intersection test, since every fragment is essentially a ray-volume intersection. Also, we don't have to test every pixel with a ray-volume test to many volumes. Only fragments can be pixels where rays intersect volumes, and every fragment is associated to only one bounding volume.

This causes another big difference between CPU and GPU implementations. In the last case, performance is not tied to subdivision as strongly as in CPU implementation. Figure 4.5 shows the framerate performance of the GPU implementation depending on the subdivision level.

As we can see, framerate is tied to the number of bounding volumes sent to the GPU, but have a different relationship than in CPU. The rendering time is related both to the number of fragments rendered and the use of the bus bandwidth, although for few Bézier patches the bus bandwidth use is negligible. An exponential growth of the Bézier patches causes an exponential growth of the bus bandwidth use, but not such a big growth of fragments. For example, Fig. 4.5 shows that sending four subpatches for the Bézier patch increments the

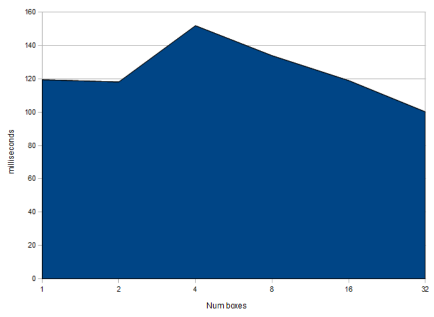


Figure 4.5: Framerate depending on the number of subdivisions.

performance. In that case, the bounding volumes adjust better to the Bézier patch leading to render fewer fragments. Despite of that, subdividing increments the number of rendered fragments since bounding volumes intersect between them. This means that in some occasions multiple fragments are rendered for the same pixel occluded by the nearest. As the number of fragments is related with the framerate, subdivision causes the technique to slow down for this reason also.

In terms of visual correctness, there is no difference between CPU and GPU implementations since they are essentially equivalent.

### 4.2.1 Bounding volumes

In this subsection we compare the two bounding volumes implemented for GPU: AABB and triangular prisms. In order to compare them better, tests calculate the framerate instead of the rendering time. Tests calculate the mean of a hundred frames in every camera position, for a hundred camera positions and we now label ray-box intersections as *Box fragments* and ray-patch intersections as *Rendered fragments*. As ray tracing is done for every fragment, the concepts are equivalent.

In Tab. 4.4 we show the comparison between performances using AABB and triangular prisms bounding volumes.

| **Statistics** | **Num patches** | **Num sub-patches** | **Box fragments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|
| AABB | 1 | 1 | 409695 | 114413 | 79.5858 |
| | 1 | 4 | 281531 | 148333 | 104.596 |
| | 1 | 16 | 221826 | 155761 | 105.324 |
| Prism | 1 | 1 | 206460 | 112325 | 119.529 |
| | 1 | 4 | 145710 | 124666 | 152.55 |
| | 1 | 16 | 124838 | 120057 | 134.979 |

Table 4.4: Comparison between triangular prisms and AABB bounding volumes

From the table we can see that triangular prisms are clearly better than AABB. As in general triangular prisms adjust better to Bézier surfaces, they produce less fragments, so they increase the framerate. We can also see that subdivision produces less occluded fragments with triangular prisms, which explains why using these bounding volumes we render less fragments. Because they adjust differently, the relationship between subdivision level and framerate changes. Figure 4.6 shows how the number of fragments decreases the more subdivided the Bézier patch is.

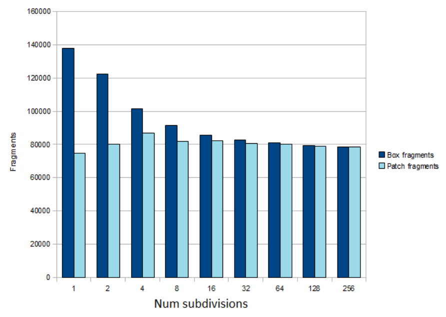


Figure 4.6: Fragments depending on the number of subdivisions.

Figure 4.7 shows the impact of the subdivision related to the framerate. As we can see, for low levels of subdivision, incrementing the subdivision level rises the performance. This is due to reducing the number of fragments. Nevertheless, incrementing too much the number of subdivisions causes a slowdown due to the rise of the geometry and other information sent. This rise causes a higher use of the graphics hardware and the bus bandwidth, slowing the execution down progressively along with the number of bounding volumes sent. As all Bézier patches and subpatches are divided in half in every subdivision level, the number of subpatches grows exponentially, which is reflected in the performance. Figures 4.6 and 4.7 show that reducing drastically the number of fragments compensates the rise of information sent. On the other hand, when the subdivision level grows high, the bounding volumes already adjusted well, causing a low reduction of fragments. This low reduction cannot compensate the rise of the information sent, and the whole system slows down.

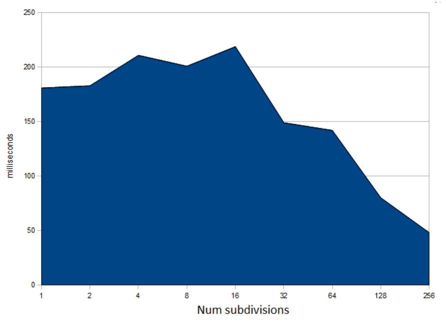


Figure 4.7: Framerate depending on the number of subdivisions.

From this point on, we always use triangle prisms for all results as they are much better bounding volumes than AABB.

## 4.2.2 Optimizations

The first optimization used is back face culling (BFC). Table 4.5 shows a comparison of the performance using or not BFC. This optimization offered by OpenGL consists in discarding the back faces of every rendered triangle. This means discarding many fragments before they get to the fragment shader. We can use this OpenGL optimization since there are no fragments generated by back faces that are not generated with front faces. Because the performance of our approach depends much on the number of fragments, BFC can speed up the rendering process. As expected, the increment in rendering speed is notorious. In our tests, we achieve approximately a 40% performance boost.

| **Statistics** | **Num patches** | **Num sub-patches** | **Box frag-ments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|
| | 1 | 1 | 206460 | 112325 | 119.529 |
| No BFC | 1 | 4 | 145710 | 124666 | 152.55 |
| | 1 | 16 | 124838 | 120057 | 134.979 |
| | 1 | 1 | 137966 | 74833.4 | 167.792 |
| BFC | 1 | 4 | 101338 | 86807.4 | 198.084 |
| | 1 | 16 | 85700.9 | 82378.7 | 203.572 |

Table 4.5: Performance comparison using BFC or not.

Our second optimization consists in using MADs to achieve an increment in rendering speed. Table 4.6 shows a comparison between shaders, one without using MADs and the other one using them. As we can see, we achieve an increment of around 10% .

| **Statistics** | **Num patches** | **Num sub-patches** | **Box frag-ments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|
| | 1 | 1 | 137966 | 74833.4 | 167.792 |
| No MAD | 1 | 4 | 101338 | 86807.4 | 198.084 |
| | 1 | 16 | 85700.9 | 82378.7 | 203.572 |
| | 1 | 1 | 137966 | 74833.4 | 181.933 |
| MAD | 1 | 4 | 101338 | 86807.4 | 211.781 |
| | 1 | 16 | 85700.9 | 82378.7 | 219.321 |

Table 4.6: Performance comparison using MAD or not.

Now, we can compare the first implementation with AABB and no optimizations with the best we have achieved. Table 4.7 shows this comparison. An increment of more than 100% is achieved. As referring to visual quality, the successive improvements do not have any impact.

| **Statistics** | **Num patches** | **Num sub-patches** | **Box frag-ments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|
| | 1 | 1 | 409695 | 114413 | 79.5858 |
| Not optimized | 1 | 4 | 281531 | 148333 | 104.596 |
| | 1 | 16 | 221826 | 155761 | 105.324 |
| | 1 | 1 | 137966 | 74833.4 | 181.933 |
| Optimized | 1 | 4 | 101338 | 86807.4 | 211.781 |
| | 1 | 16 | 85700.9 | 82378.7 | 219.321 |

Table 4.7: Comparison between the first implementation and the most optimized version

### 4.2.3 Performance

As done in previous section with the CPU implementation, we need to choose the parameters that optimize the performance of our approach without compromising visual correctness. Table 4.8 shows different tests executed different parameters. From these results and the ones explained in previous subsections, we conclude that 4 iterations and 16 subpatches is the best option.

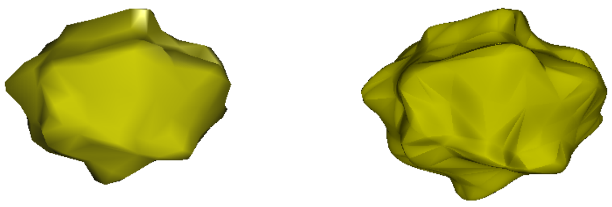
| **Statistics** | **Num patches** | **Num sub-patches** | **Box frag-ments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|
| 4 iterations | 1 | 1 | 137966 | 74833.4 | 181.933 |
| | 1 | 4 | 101338 | 86807.4 | 211.781 |
| | 1 | 16 | 85700.9 | 82378.7 | 219.321 |
| 6 iterations | 1 | 1 | 137966 | 74987.9 | 149.618 |
| | 1 | 4 | 101338 | 86842 | 177.555 |
| | 1 | 16 | 85700.9 | 82388.9 | 183.354 |
| 8 iterations | 1 | 1 | 137966 | 75021.2 | 121.75 |
| | 1 | 4 | 101338 | 86851.3 | 147.611 |
| | 1 | 16 | 85700.9 | 82390.7 | 157.357 |

Table 4.8: Performance related with the number of Newton iterations

## 4.3 GPU ray tracing a PN Triangle mesh

In this section, we show the results of the implementation of our approach rendering two different 3D models. The first model represents a rock with 500 faces and the second one represents a horse with 96966 faces.

Figure 4.11(a) shows the rock rendered as a triangle mesh and as PN triangles.

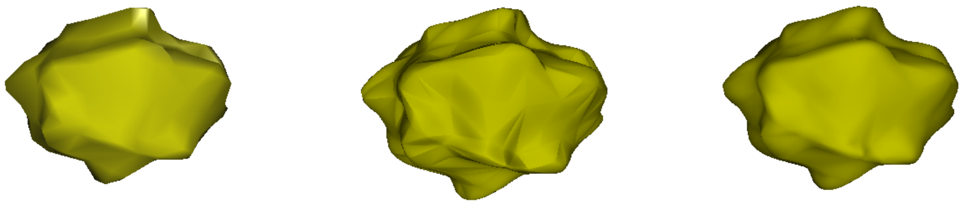


(a) Triangle mesh. (b) PN triangles.

Figure 4.8: Comparison between techniques.

As we can see, the PN triangles version does not give a very good visual quality because of the normals discontinuity. For this reason we implemented the curved PN triangles rendering. Figure 4.9 shows the difference between the three methods. Curved PN triangles offer a much better quality because of the normal continuity and smoothness.

Table 4.9 shows the different performances using the three techniques. From the results in this table, we can conclude that in performance terms, the difference between rendering PN



(a) Triangle mesh. (b) PN triangles. (c) Curved PN triangles.

Figure 4.9: Comparison between techniques.

triangles and curved PN triangles is negligible. We can also observe that our approach can render 3D objects using curved PN triangles with high interactive framerates.

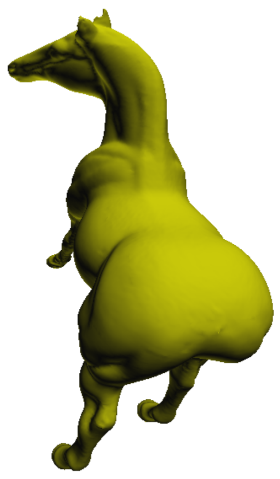
| **Statistics** | **Num patches** | **Num sub-patches** | **Subdivision level** | **Box frag-ments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|---|
| Triangle mesh | 500 | - | - | - | 41535.4 | 521.849 |
| PN triangles | 500 | 500 | 0 | 86396.7 | 59873.6 | 138.141 |
| | 500 | 2000 | 2 | 78059.8 | 67708.8 | 115.255 |
| | 500 | 8000 | 4 | 70644.3 | 67522.3 | 51.7642 |
| Curved PN triangles | 500 | 500 | 0 | 86302.2 | 59813.9 | 140.358 |
| | 500 | 2000 | 2 | 78059.8 | 67708.9 | 110.622 |
| | 500 | 8000 | 4 | 70644.3 | 67522.2 | 47.1471 |

Table 4.9: Performance comparison rendering the rock 3D model

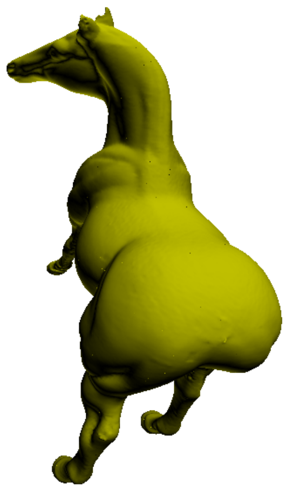
In Tab. 4.9 we also show the dependence of the framerate to the subdivision level for every technique. There is no subdivision when rendering triangle meshes and, consequently, the rendering speed is independent. This is not the case when rendering Bézier patches. Every subdivision level divides patches into two subpatches, so the number of patches is multiplied by the same number. This time, as every Bézier patch is rendered in a small number of pixels, incrementing the subdivision level does not cause a major drop in the number of fragments. In fact, incrementing the number of subpatches can even increase the number of fragments processed by fragment shader since many of them can overlap in one pixel. Augmenting the number of bounding volumes, augments also the number of overlapped fragments. This leads to a slower performance because the number of processed fragments in the frame does not drop.

The horse model is a high detail model, which means that rendering it is a time consuming task. In order to use our technique there is no need for such detail because curved PN triangles are smooth both in normal continuity and geometrically in the silhouette. Therefore, we created a low detail version of the horse triangle mesh and performed all tests with it. Figure 4.10 shows the difference of the results rendering curved PN triangles generated from high and low resulution versions of the model.

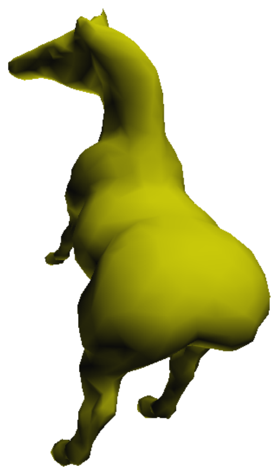
In visual correctness terms, to render correctly we use the parameters and results obtained in previous section. This way, Fig. 4.11 is the rendering of both models with four Newton's iterations and four subdivision levels. As we can see in the figure, some artifacts are visible producing holes in the models. This means that we need to optimize our technique to eliminate the artifacts and achieve correct results, which is what we discuss in next subsection.



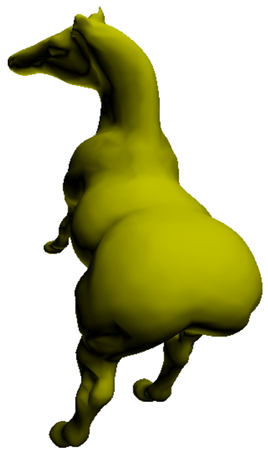
(a) High detail triangle mesh.



(b) Curved PN triangles generated by high detail mesh.



(c) Low detail triangle mesh.



(d) Curved PN triangles generated by low detail mesh.

Figure 4.10: Horse rendered using triangle mesh and Bézier patches.

### 4.3.1 Optimizations

Since the goal is to achieve correct images, there are essentially two solutions: subdividing only when needed, and changing the parameters. Both solutions are compatible and we have implemented them. In this subsection we will have a look at the first one and in the next subsection the second one.

Very deformed Bézier patches are more likely to produce errors in rendering than planar ones. Therefore, identifying the most triangle-like patches allows us to stop subdividing them. Table 4.10 shows the impact of this optimization.

And for the horse, Tab. 4.11 shows the results.

As we can see, it is an easy optimization with few changes on the code that has no impact

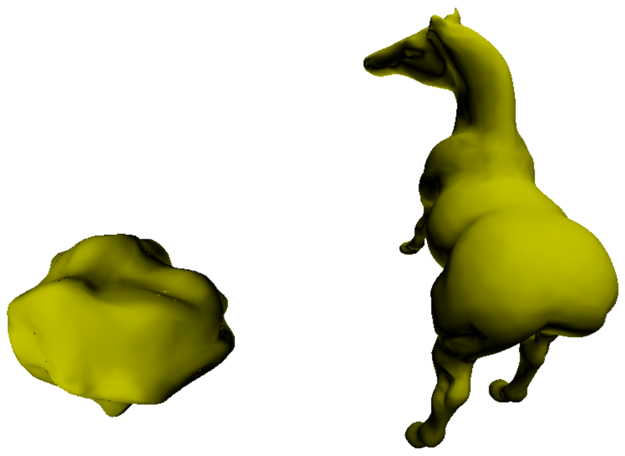

(a) Rock model. (b) Horse model.

Figure 4.11: 3D models rendered using our approach.

| **Rock** | **Num patches** | **Num sub-patches** | **Subdivision level** | **Box fragments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|---|
| Totally subdivided | 500 | 500 | 0 | 86302.2 | 59813.9 | 140.358 |
| | 500 | 2000 | 2 | 78059.8 | 67708.9 | 110.622 |
| | 500 | 8000 | 4 | 70644.3 | 67522.2 | 47.1471 |
| Partially subdivided | 500 | 500 | 0 | 86302.2 | 59813.9 | 140.358 |
| | 500 | 1970 | 2 | 78097.7 | 67664.5 | 115.851 |
| | 500 | 6503 | 4 | 71035.1 | 67389.3 | 52.356 |

Table 4.10: Performance comparison rendering the rock 3D model subdividing totally and partially

| **Horse** | **Num patches** | **Num sub-patches** | **Subdivision level** | **Box fragments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|---|
| Totally subdivided | 2000 | 2000 | 0 | 48711.2 | 35803.9 | 100.735 |
| | 2000 | 8000 | 2 | 43120.7 | 38281.6 | 42.6333 |
| | 2000 | 32000 | 4 | 38864.2 | 37546.9 | 13.2073 |
| Partially subdivided | 2000 | 2000 | 0 | 48711.2 | 35803.9 | 100.735 |
| | 2000 | 7644 | 2 | 43073.7 | 38020.9 | 44.0313 |
| | 2000 | 21964 | 4 | 32102.3 | 30408.3 | 18.4815 |

Table 4.11: Performance comparison rendering the horse 3D model subdividing totally and partially

on visual correctness ans it has an impact on speed. Although low or nonexistent for low levels of subdivision, for greater ones the impact of the optimization increases. From this point on, we will always use partial subdivision.

The second optimization consists in using the idea of multiple seeds. This way, we have four seeds: the same as in previous versions and three more, one for each vertex of the triangular Bézier patch. As we can see in Tab. 4.12 and 4.13 multiple seeds optimization is much more stable to subdivision in framerate terms. This happens because the number of rendered fragments is also more stable, and as the fragment shader has a bigger load, it is much more fragment limited.

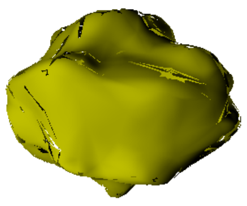
| **Rock** | **Num patches** | **Num sub-patches** | **Subdivision level** | **Box fragments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|---|
| | 500 | 500 | 0 | 86302.2 | 59813.9 | 140.358 |
| Single | 500 | 1970 | 2 | 78097.7 | 67664.5 | 115.851 |
| seed | 500 | 6503 | 4 | 71035.1 | 67389.3 | 52.356 |
| | 500 | 500 | 0 | 86396.7 | 62282.5 | 59.6103 |
| Multiple | 500 | 1970 | 2 | 78097.7 | 69522.2 | 55.1245 |
| seeds | 500 | 6503 | 4 | 71035.1 | 68105.4 | 39.4463 |

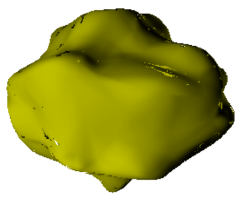
Table 4.12: Performance comparison rendering the rock 3D model using single and multiple seeds

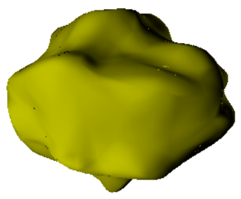
| **Horse** | **Num patches** | **Num sub-patches** | **Subdivision level** | **Box fragments** | **Rendered fragments** | **FPS** |
|---|---|---|---|---|---|---|
| | 2000 | 2000 | 0 | 48711.2 | 35803.9 | 100.735 |
| Single | 2000 | 7644 | 2 | 43073.7 | 38020.9 | 44.0313 |
| seed | 2000 | 21964 | 4 | 32102.3 | 30408.3 | 18.4815 |
| | 2000 | 2000 | 0 | 48429.2 | 35146 | 58.844 |
| Multiple | 2000 | 7644 | 2 | 43073.7 | 38238.3 | 33.8689 |
| seeds | 2000 | 21964 | 4 | 39611.2 | 37507.2 | 17.3512 |

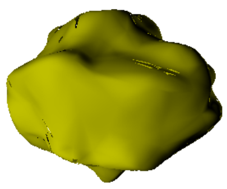
Table 4.13: Performance comparison rendering the horse 3D model using single and multiple seeds

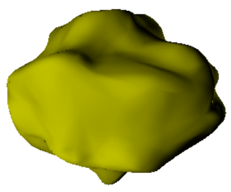
Although rendering with multiple seeds performs slower, the visual correctness is much better. Figure 4.12 shows the different results obtained changing the number of subdivisions using single and multiple seeds. As we can see, with multiple seeds we obtain correct images with only two levels of subdivision whereas with only one seed we do not achieve them even with four levels of subdivision. This means that multiple seeds is both an optimization in correctness and in speed, taking into account Tab. 4.12.


(a) Single seed and no subdivision.


(b) Single seed and 2 subdivision levels.


(c) Single seed and 4 subdivision levels.


(d) Multiple seeds and no subdivision.


(e) Multiple seeds and 2 subdivision levels.

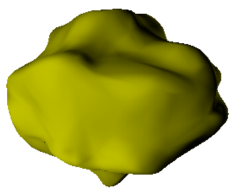

(f) Multiple seeds and 4 subdivision levels.

Figure 4.12: Comparison between single and multiple seeds.

CHAPTER 5

# Conclusions

In this chapter we discuss the work explained in previous chapters. In the first section, we summarize the main contributions in our research, and state some conclusions about the obtained results, comparing them with what we expected. The second section is a list of future work to improve the quality of present results.

## 5.1 Thesis Contributions

The main contribution of this Thesis is adapting Newton's method to calculate intersections between a ray and a triangular Bézier patch in an efficient GPU implementation. Calculating the intersections in the fragment shader speeds up our approach thanks to the intensive use of the graphics hardware.

In our approach we have adapted the ideas described in articles [2] and [5] to be able to implement them in efficient fragment shaders that renders curved PN triangles. Our results prove that our approach is much faster. In the results chapter we have seen that our approach is able to render 3D models within interactive framerates converting the triangle mesh into curved PN triangles. In contrast with CPU version, GPU approach can render some orders of magnitude faster.

Even though the intensive use of graphics hardware already achieves very good results, it is a hardware optimized for rendering triangles. This means that if we had an optimized hardware for our approach we would be able to render much quicker. For example, Newton's iterations could be executed in hardware and multiple seeds optimization could be parallelized.

Also, our tests were executed in a laptop with a slow graphics card. We could get much better framerates with a newer one, especially if early depth test is available. This optimization would drastically reduce the number of processed fragments and thus, boosting the framerate.

In conclusion, our approach has proven efficient and could be used in many scenarios. The adaptation from any existing graphical software should be quick since few changes would be needed. Our software automatically converts triangle meshes in *ply* format into lists of triangular Bézier surfaces and the shader that renders them is already implemented. Therefore, the integration of our technique into another graphics software consists only in sending the geometry and information generated by our file reader and including our fragment shader correctly initialized.

In comparison with the expected, the results are satisfactory. We are able to render objects with infinite detail in real time, and there no or very little artifacts seen.

## 5.2 Future work

Our approach provides $u$ and $v$ coordinates as a side result of the ray casting process. We could easily use those coordinates to implement many texture techniques such as texture mapping, normal mapping or parallax mapping to increment realism and detail with few rendering time overhead.

Another side result of our approach is the first and second derivatives. With these we could obtain information about the curvature in each point. This could be used to implement a simple version of an ambient occlusion, providing a much better illumination. As the most part of the calculations are already done, the impact in performance should be low, whereas the visual quality should improve notably.

Combining those techniques should provide high visual quality renders with similar performances and few changes in the code.

# APPENDIX A

# Shaders

## A.1 Vertex Shader

```
void main(void)
{
   gl_FrontColor = gl_Color;

   gl_Position = ftransform();

}
```

Code A.1: Vertex shader

## A.2 Fragment Shader

### A.2.1 Single seed

```
/* Fragment shader */

uniform vec3 minp;      // Bounding volume minimum coordinates
uniform vec3 maxp;      // Bounding volume maximum coordinates
uniform vec3 obs;    // Observers coordinates
uniform vec4 color;

//Constants of the Bezier patch
uniform vec3 Q30;
uniform vec3 Q03;
uniform vec3 Q21;
uniform vec3 Q12;
uniform vec3 Q20;
uniform vec3 Q02;
uniform vec3 Q11;
uniform vec3 Q10;
uniform vec3 Q01;
uniform vec3 Q00;

//Constants of the normal patch
uniform vec3 n200;
uniform vec3 n020;
uniform vec3 n002;
uniform vec3 n110;
uniform vec3 n011;
uniform vec3 n101;

//Seeds
uniform vec2 nas;
uniform vec2 nas1;
uniform vec2 nas2;
uniform vec2 nas3;

//Precision
uniform float error;


void main()
{

      // Fragment coordinates in scene coordinates
      vec3 pos=maxp-minp;
      pos.xyz = gl_Color.rgb*(pos.xyz)+minp.xyz;

      vec3 ray = pos-obs;

      // Planes P1 and P2
      vec3 N1=cross(ray,vec3(-1.0));
      vec3 N2=cross(ray,N1);
      float d1=-dot(N1,obs);
      float d2=-dot(N2,obs);

      // Variables declarations
      vec2 res=vec2(nas.x,nas.y);
      mat2 inverseJacob;
      vec2 R;
      vec3 dBu;
      vec3 dBv;
      vec3 Buv;
      float inverseConstant;
      vec3 aux;

      float u;
      float v;

      for(int i=0;i<4;i++){

         u=res.x;
         v=res.y;

         // Partial derivative of B by u
         dBu=Q21*v + Q20;
         aux=Q30*u*3.0;
         dBu=2.0*dBu + aux;
         aux=Q12*v + Q11;
         aux=v*aux + Q10;
         dBu=dBu*u + aux;

         // Partial derivative of B by v
```

```
            aux=Q12*u + Q02;
            dBv=Q03*3.0*v;
            dBv=2.0*aux + dBv;
            aux=Q21*u + Q11;
            aux=aux*u + Q01;
            dBv=v*dBv + aux;

            // Derivative of B
            Buv=Q21*v + Q20;
            Buv=Q30*u + Buv;
            aux=Q12*v + Q11;
            aux=aux*v + Q10;
            Buv=u*Buv + aux;
            aux=Q03*v + Q02;
            aux=v*aux + Q01;
            aux=v*aux + Q00;
            Buv=u*Buv + aux;

            R= vec2(dot(N1,Buv)+d1 , dot(N2,Buv)+d2);

            float dotN1Bu=dot(N1,dBu);
            float dotN1Bv=dot(N1,dBv);
            float dotN2Bu=dot(N2,dBu);
            float dotN2Bv=dot(N2,dBv);
            inverseConstant=(1.0/ ( dotN1Bu*dotN2Bv - dotN1Bv*dotN2Bu ) );

            inverseJacob=mat2(   dotN2Bv*inverseConstant , -dotN2Bu*
                inverseConstant,
                              -dotN1Bv*inverseConstant, dotN1Bu*
                                  inverseConstant);

            // Newton's iteration
            res= res - inverseJacob*R;

         }

         float w=1.0-res.x-res.y;

         // Condition to discard this seed
         if(res.x>=-0.0001 && res.x<=1.0001 && res.y>=-0.0001 && res.y
             <=1.0001 && w>=-0.0001 && abs(dot(N1, Buv)+d1)<error && abs(
             dot(N2, Buv)+d2)<error){

            vec3 N=n200*res.x*res.x + n020*res.y*res.y + n002*w*w + n110*
                res.x*res.y + n011*res.y*w + n101*res.x*w;
            vec3 l2=vec3(0.0,0.0,-1.0);
            float lconst=abs(dot(normalize(N),l2));
            gl_FragColor = vec4(lconst*color.x,lconst*color.y,lconst*color.
                z,color.w);
         }
         // Discarding the fragment
         else
            discard;




}
```

Code A.2: Fragment shader

### A.2.2 Multiple seeds

```
/* Fragment shader */

uniform vec3 minp;      // Bounding volume minimum coordinates
uniform vec3 maxp;      // Bounding volume maximum coordinates
uniform vec3 obs;    // Observers coordinates
uniform vec4 color;

//Constants of the Bezier patch
uniform vec3 Q30;
uniform vec3 Q03;
uniform vec3 Q21;
uniform vec3 Q12;
uniform vec3 Q20;
uniform vec3 Q02;
uniform vec3 Q11;
uniform vec3 Q10;
uniform vec3 Q01;
uniform vec3 Q00;

//Constants of the normal patch
uniform vec3 n200;
uniform vec3 n020;
uniform vec3 n002;
uniform vec3 n110;
uniform vec3 n011;
uniform vec3 n101;

//Seeds
uniform vec2 nas;
uniform vec2 nas1;
uniform vec2 nas2;
uniform vec2 nas3;

//Precision
uniform float error;


void main()
{

      // Fragment coordinates in scene coordinates
      vec3 pos=maxp-minp;
      pos.xyz = gl_Color.rgb*(pos.xyz)+minp.xyz;

      vec3 ray = pos-obs;

      // Planes P1 and P2
      vec3 N1=cross(ray,vec3(-1.0));
      vec3 N2=cross(ray,N1);
      float d1=-dot(N1,obs);
      float d2=-dot(N2,obs);

      // Variables declarations
      vec2 res=vec2(nas.x,nas.y);
      mat2 inverseJacob;
      vec2 R;
      vec3 dBu;
      vec3 dBv;
      vec3 Buv;
      float inverseConstant;
      vec3 aux;

      float u;
      float v;

      for(int i=0;i<4;i++){

         u=res.x;
         v=res.y;

         // Partial derivative of B by u
         dBu=Q21*v + Q20;
         aux=Q30*u*3.0;
         dBu=2.0*dBu + aux;
         aux=Q12*v + Q11;
         aux=v*aux + Q10;
         dBu=dBu*u + aux;

         // Partial derivative of B by v
         aux=Q12*u + Q02;
         dBv=Q03*3.0*v;
         dBv=2.0*aux + dBv;
         aux=Q21*u + Q11;
         aux=aux*u + Q01;
         dBv=v*dBv + aux;

         // Derivative of B
         Buv=Q21*v + Q20;
         Buv=Q30*u + Buv;
         aux=Q12*v + Q11;
         aux=aux*v + Q10;
         Buv=u*Buv + aux;
         aux=Q03*v + Q02;
         aux=v*aux + Q01;
         aux=v*aux + Q00;
         Buv=u*Buv + aux;

         R= vec2(dot(N1,Buv)+d1 , dot(N2,Buv)+d2);

         float dotN1Bu=dot(N1,dBu);
         float dotN1Bv=dot(N1,dBv);
         float dotN2Bu=dot(N2,dBu);
         float dotN2Bv=dot(N2,dBv);
         inverseConstant=(1.0/ ( dotN1Bu*dotN2Bv - dotN1Bv*dotN2Bu ) );

         inverseJacob=mat2(   dotN2Bv*inverseConstant , -dotN2Bu*
             inverseConstant,
                            -dotN1Bv*inverseConstant, dotN1Bu*
                                inverseConstant);

         // Newton's iteration
         res= res - inverseJacob*R;

      }

      float w=1.0-res.x-res.y;

      // Condition to discard this seed
      if(res.x>=-0.0001 && res.x<=1.0001 && res.y>=-0.0001 && res.y
          <=1.0001 && w>=-0.0001 && abs(dot(N1, Buv)+d1)<error && abs(
          dot(N2, Buv)+d2)<error){
            vec3 N=n200*res.x*res.x + n020*res.y*res.y + n002*w*w + n110*
                res.x*res.y + n011*res.y*w + n101*res.x*w;
            vec3 l2=vec3(0.0,0.0,-1.0);
            float lconst=abs(dot(normalize(N),l2));
            gl_FragColor = vec4(lconst*color.x,lconst*color.y,lconst*color.
                z,color.w);
        }
        // Repeating the process with another seed
        else{

            vec2 res=vec2(nas1.x,nas1.y);
            for(int i=0;i<4;i++){

                u=res.x;
                v=res.y;
                dBu=Q21*v + Q20;
                aux=Q30*u*3.0;
                dBu=2.0*dBu + aux;
                aux=Q12*v + Q11;
                aux=v*aux + Q10;
                dBu=dBu*u + aux;

                aux=Q12*u + Q02;
                dBv=Q03*3.0*v;
                dBv=2.0*aux + dBv;
                aux=Q21*u + Q11;
                aux=aux*u + Q01;
                dBv=v*dBv + aux;

                Buv=Q21*v + Q20;
                Buv=Q30*u + Buv;
                aux=Q12*v + Q11;
                aux=aux*v + Q10;
                Buv=u*Buv + aux;
                aux=Q03*v + Q02;
                aux=v*aux + Q01;
                aux=v*aux + Q00;
                Buv=u*Buv + aux;

                R= vec2(dot(N1,Buv)+d1 , dot(N2,Buv)+d2);

                float dotN1Bu=dot(N1,dBu);
                float dotN1Bv=dot(N1,dBv);
                float dotN2Bu=dot(N2,dBu);
                float dotN2Bv=dot(N2,dBv);
                inverseConstant=(1.0/ ( dotN1Bu*dotN2Bv - dotN1Bv*dotN2Bu )
                    );

                inverseJacob=mat2(   dotN2Bv*inverseConstant , -dotN2Bu*
                    inverseConstant,
                                  -dotN1Bv*inverseConstant, dotN1Bu*
                                      inverseConstant);

                res= res - inverseJacob*R;

            }

            float w=1.0-res.x-res.y;

            if(res.x>=-0.0001 && res.x<=1.0001 && res.y>=-0.0001 && res.y
                <=1.0001 && w>=-0.0001 && abs(dot(N1, Buv)+d1)<error && abs
                (dot(N2, Buv)+d2)<error){
```

```
            vec3 N=n200*res.x*res.x + n020*res.y*res.y + n002*w*w + n110
                *res.x*res.y + n011*res.y*w + n101*res.x*w;
            vec3 l2=vec3(0.0,0.0,-1.0);
            float lconst=abs(dot(normalize(N),l2));
            gl_FragColor = vec4(lconst*color.x,lconst*color.y,lconst*
                color.z,color.w);
        }
        else{

            vec2 res=vec2(nas2.x,nas2.y);
            for(int i=0;i<4;i++){

                u=res.x;
                v=res.y;
                dBu=Q21*v + Q20;
                aux=Q30*u*3.0;
                dBu=2.0*dBu + aux;
                aux=Q12*v + Q11;
                aux=v*aux + Q10;
                dBu=dBu*u + aux;

                aux=Q12*u + Q02;
                dBv=Q03*3.0*v;
                dBv=2.0*aux + dBv;
                aux=Q21*u + Q11;
                aux=aux*u + Q01;
                dBv=v*dBv + aux;

                Buv=Q21*v + Q20;
                Buv=Q30*u + Buv;
                aux=Q12*v + Q11;
                aux=aux*v + Q10;
                Buv=u*Buv + aux;
                aux=Q03*v + Q02;
                aux=v*aux + Q01;
                aux=v*aux + Q00;
                Buv=u*Buv + aux;

                R= vec2(dot(N1,Buv)+d1 , dot(N2,Buv)+d2);

                float dotN1Bu=dot(N1,dBu);
                float dotN1Bv=dot(N1,dBv);
                float dotN2Bu=dot(N2,dBu);
                float dotN2Bv=dot(N2,dBv);
                inverseConstant=(1.0/ ( dotN1Bu*dotN2Bv - dotN1Bv*dotN2Bu
                     ) );

                inverseJacob=mat2(   dotN2Bv*inverseConstant , -dotN2Bu*
                    inverseConstant,
                                  -dotN1Bv*inverseConstant, dotN1Bu*
                                      inverseConstant);

                res= res - inverseJacob*R;

            }

            float w=1.0-res.x-res.y;

            if(res.x>=-0.0001 && res.x<=1.0001 && res.y>=-0.0001 && res.
                y<=1.0001 && w>=-0.0001 && abs(dot(N1, Buv)+d1)<error &&
                 abs(dot(N2, Buv)+d2)<error){
```

```
                vec3 N=n200*res.x*res.x + n020*res.y*res.y + n002*w*w +
                    n110*res.x*res.y + n011*res.y*w + n101*res.x*w;
                vec3 l2=vec3(0.0,0.0,-1.0);
                float lconst=abs(dot(normalize(N),l2));
                gl_FragColor = vec4(lconst*color.x,lconst*color.y,lconst*
                    color.z,color.w);
             }
             else{
                vec2 res=vec2(nas3.x,nas3.y);
                for(int i=0;i<4;i++){

                   u=res.x;
                   v=res.y;
                   dBu=Q21*v + Q20;
                   aux=Q30*u*3.0;
                   dBu=2.0*dBu + aux;
                   aux=Q12*v + Q11;
                   aux=v*aux + Q10;
                   dBu=dBu*u + aux;

                   aux=Q12*u + Q02;
                   dBv=Q03*3.0*v;
                   dBv=2.0*aux + dBv;
                   aux=Q21*u + Q11;
                   aux=aux*u + Q01;
                   dBv=v*dBv + aux;

                   Buv=Q21*v + Q20;
                   Buv=Q30*u + Buv;
                   aux=Q12*v + Q11;
                   aux=aux*v + Q10;
                   Buv=u*Buv + aux;
                   aux=Q03*v + Q02;
                   aux=v*aux + Q01;
                   aux=v*aux + Q00;
                   Buv=u*Buv + aux;

                   R= vec2(dot(N1,Buv)+d1 , dot(N2,Buv)+d2);

                   float dotN1Bu=dot(N1,dBu);
                   float dotN1Bv=dot(N1,dBv);
                   float dotN2Bu=dot(N2,dBu);
                   float dotN2Bv=dot(N2,dBv);
                   inverseConstant=(1.0/ ( dotN1Bu*dotN2Bv - dotN1Bv*
                       dotN2Bu ) );

                   inverseJacob=mat2(   dotN2Bv*inverseConstant , -
                       dotN2Bu*inverseConstant,
                                     -dotN1Bv*inverseConstant, dotN1Bu*
                                         inverseConstant);

                   res= res - inverseJacob*R;

                }

                float w=1.0-res.x-res.y;

                if(res.x>=-0.0001 && res.x<=1.0001 && res.y>=-0.0001 &&
                    res.y<=1.0001 && w>=-0.0001 && abs(dot(N1, Buv)+d1)<
                    error && abs(dot(N2, Buv)+d2)<error){
```

```
                vec3 N=n200*res.x*res.x + n020*res.y*res.y + n002*w*w
                    + n110*res.x*res.y + n011*res.y*w + n101*res.x*w;
                vec3 l2=vec3(0.0,0.0,-1.0);
                float lconst=abs(dot(normalize(N),l2));
                gl_FragColor = vec4(lconst*color.x,lconst*color.y,
                    lconst*color.z,color.w);
            }
            // All seeds have been tried, we discard the fragment
            else {
                discard;
            }
        }
    }
}
}
```

Code A.3: Fragment shader

# Bibliography

[1] Unigine, http://unigine.com/.

[2] Abert, O., Geimer, M., and Muller, S. Direct and fast ray tracing of nurbs surfaces. *Symposium on Interactive Ray Tracing 0* (2006), 161–168.

[3] Boubekeur, T., and Schlick, C. Generic mesh refinement on gpu. In *ACM SIGGRAPH/Eurographics Graphics Hardware* (2005).

[4] Cook, R. L., Carpenter, L., and Catmull, E. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.* (1987), 95–102.

[5] Geimer, M., and Abert, O. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *In Proceedings of WSCG* (2005), pp. 71–78.

[6] Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., and Tachi, S. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001* (2001), pp. 205–208.

[7] McGuire, M., and McGuire, M. Steep parallax mapping. *I3D 2005 Poster* (April 2005).

[8] Oetiker, T., Partl, H., Hyna, I., and Schlegl, E. *The Not So Short Introduction to LaTeX.* Addison-Wesley Professional, June 2007. Version 4.22.

[9] Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. Curved pn triangles. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), ACM, pp. 159–166.

[10] Wald, I., Benthin, C., and Slusallek, P. Distributed interactive ray tracing of dynamic scenes. *Parallel and Large-Data Visualization and Graphics, IEEE Symposium on 0* (2003), 11.

[11] Wald, I., Slusallek, P., Benthin, C., and Wagner, M. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* (2001), pp. 153–164.

[12] Woo, A. *Fast ray-box intersection.* Academic Press Professional, Inc., San Diego, CA, USA, 1990.