Ray Tracing Non-Polygonal Objects: Implementation and Performance Analysis using Embree

Mälardalens University
School of Innovation, Design and Engineering
Michael Carlie
Bachelor Thesis in Computer Science
2016-06-15
Examiner: Thomas Larsson
Supervisor: Afshin Ameri
Abstract

Free-form surfaces and implicit surfaces must be tessellated before being rendered with rasterization techniques. However ray tracing provides the means to directly render such objects without the need to first convert into polygonal meshes. Since ray tracing can handle triangle meshes as well, the question of which method is most suitable in terms of performance, quality and memory usage is addressed in this thesis.

Bézier surfaces and NURBS surfaces along with basic algebraic implicit surfaces are implemented in order to test the performance relative to polygonal meshes approximating the same objects. The parametric surfaces are implemented using an iterative Newtonian method that converges on the point of intersection using a bounding volume hierarchy that stores the initial guesses. Research into intersecting rays with parametric surfaces is surveyed in order to find additional methods that speed up the computation. The implicit surfaces are implemented using common direct algebraic methods. All of the intersection tests are implemented using the Embree ray tracing API as well as a SIMD library in order to achieve interactive framerates on a CPU.

The results show that both Bézier surfaces and NURBS surfaces can achieve interactive framerates on a CPU using SIMD computation, with Bézier surfaces coming close to the performance of polygonal counterparts. The implicit surfaces implemented outperform even the simplest polygonal approximations.
Table of Contents

Abstract ................................................................................................................................. 2

1. Introduction .................................................................................................................... 5
   1.1 Problem Definition ...................................................................................................... 5

2. Background ...................................................................................................................... 5
   2.1 Rendering Algorithms ............................................................................................... 5
   2.2 Ray Tracing ............................................................................................................... 6
      2.2.1 Illumination .......................................................................................................... 7
   2.3 Bounding Volume Hierarchy .................................................................................... 8
   2.4 Embree .................................................................................................................... 8
   2.5 Non-Polygonal Ray Tracing ....................................................................................... 11
      2.5.1 Ray Tracing Implicit Surfaces ............................................................................ 11
      2.5.2 Ray Tracing Parametric Surfaces ........................................................................ 12

3. Method ............................................................................................................................. 13

4. Theory .............................................................................................................................. 13
   4.1 Implicit Surfaces ....................................................................................................... 13
   4.2 Parametric Surfaces ............................................................................................... 14
   4.3 Continuity ................................................................................................................. 15
   4.4 Bézier Curves and Surfaces ..................................................................................... 15
      4.4.1 Bézier Curves ...................................................................................................... 15
      4.4.2 Continuity between Bézier Curves ...................................................................... 16
      4.4.3 De Casteljau’s Algorithm .................................................................................... 17
      4.4.4 Bézier Surfaces .................................................................................................. 17
      4.4.5 Partial Derivatives ............................................................................................... 18
      4.4.6 Matrix Representation ....................................................................................... 18
   4.5 Non-Uniform Rational B-Spline (NURBS) Curves and Surfaces ....................... 19
      4.5.1 NURBS Curve ..................................................................................................... 19
      4.5.2 Basis Functions .................................................................................................. 20
      4.5.3 Knot Vector ......................................................................................................... 21
      4.5.4 Local Support ...................................................................................................... 22
      4.5.5 Knot Insertion ..................................................................................................... 22
      4.5.6 NURBS Surface .................................................................................................. 24
      4.5.7 Partial Derivatives ............................................................................................... 25

5 Ray Tracing Implicit Surfaces ....................................................................................... 26
   5.1 Ray-Sphere Intersection ......................................................................................... 26
   5.2 Ray-Cylinder Intersection ....................................................................................... 27

6. Ray Tracing Parametric Surfaces .................................................................................. 28
   6.1 Newton Raphson ..................................................................................................... 28
   6.2 Ray Representation ................................................................................................... 29
   6.3 Newton’s Method for Ray-Surface Intersection ...................................................... 29
   6.4 Bézier Surface Evaluation ....................................................................................... 31
   6.5 Bézier Surface Subdivision ....................................................................................... 31
      6.5.1 Bézier Flatness Test ............................................................................................ 32
   6.6 NURBS Surface Evaluation ..................................................................................... 33
      6.6.1 Cox De-Boor Items ............................................................................................ 34
      6.6.2 Basis-Function Cache ........................................................................................ 35
   6.7 NURBS Subdivision ................................................................................................. 36
6.7.1 NURBS Flatness Test ...........................................................................................................36
7. Implementation .........................................................................................................................37
  7.1 Implicit Surfaces ....................................................................................................................37
  7.2 Parametric Surfaces ..............................................................................................................37
     7.2.1 Bezier Surface Preprocessing .......................................................................................37
     7.2.2 NURBS Surface Preprocessing ...................................................................................38
     7.2.3 Bounding volume hierarchy .........................................................................................40
     7.2.4 Intersection Testing ......................................................................................................40
     7.2.5 Bezier Surface Evaluation ............................................................................................41
     7.2.6 NURBS Surface Evaluation ..........................................................................................41
8. Results ......................................................................................................................................41
  8.1 Test System ...........................................................................................................................41
  8.2 Test Setup .............................................................................................................................41
  8.3 Polygonal Meshes ..................................................................................................................42
     8.3.1 Image Quality ...............................................................................................................42
     8.3.3 Memory Consumption .................................................................................................43
  8.4 Implicit Surfaces ...................................................................................................................43
     8.4.1 Image Quality ...............................................................................................................43
     8.4.2 Performance ..................................................................................................................44
     8.4.3 Memory Consumption .................................................................................................46
  8.5 Bezier Surfaces .....................................................................................................................46
     8.5.1 Image Quality ...............................................................................................................46
     8.5.2 Performance ..................................................................................................................47
     8.5.3 Memory Consumption .................................................................................................49
  8.6 NURBS ...................................................................................................................................49
     8.6.1 Image Quality ...............................................................................................................49
     8.6.2 Performance ..................................................................................................................50
     8.6.3 Memory Consumption .................................................................................................52
9. Discussion .................................................................................................................................52
  9.1 Related Research ...................................................................................................................53
10. Conclusions .............................................................................................................................54
    10.1 Future work .......................................................................................................................54
Bibliography ................................................................................................................................55
1. Introduction
Ray Tracing is a rendering technique that produces images by generating rays that are projected from the camera. Intersection tests are conducted between the rays and the geometry representations that constitute the scene in order to determine visibility and the amount of light that is reflected back along the rays towards the camera. Ray tracing is well suited for rendering photo-realistic images with reflections, refractions and other lighting techniques since the generated rays simulate the behavior of light. Geometry representations in rendering applications are typically represented using a set of adjacent polygonal faces that together represent the geometry. With ray tracing such polygonal meshes are rendered by conducting intersection tests with each of the polygonal faces in the mesh, however ray tracing can also be used to render non-polygonal mathematically defined geometry representations. Many real-world objects can be represented using such mathematically defined geometries. Examples include implicit surfaces such as cones, cylinders ellipsoids and spheres, as well as Bézier surfaces and NURBS. Traditionally these geometries are approximated and represented using polygonal meshes in rendering applications through a process known as tessellation, however such methods have problems with unintended artefacts and limited scalability.

1.1 Problem Definition
The ability to directly render non-polygonal geometry representations is useful for applications that need to circumvent the limitations of traditional polygonal meshes, however certain types of rendering applications, such as those that provide interactivity typically have performance constraints. Ray tracing implementations for various geometry representations have varying complexity and resource usage which limits the types of geometry representations that might be suitable for applications with specific performance requirements. With this in mind the central questions that this paper will address are:

- Which non-polygonal geometry representations are suitable for usage in real-time ray tracing and how can they be implemented?
- How do various non-polygonal geometry representations compare with each other and their tessellated counterparts in terms of performance and resource usage?

2. Background

2.1 Rendering Algorithms
The fundamental algorithms used to represent images in computer graphics can be generally classified as either projective or image-space algorithms. Projective algorithms typically use a process known as rasterization to map the three-dimensional geometric representations to pixels in a two-dimensional space in a per object fashion. The shading of the individual objects is performed on each of the objects locally in order to give the appearance of depth and determine how they interact with light. Image-space algorithms on the other hand determine the visibility and shading of an object by finding
geometries along a line of sight and computing the amount of light that is reflected back towards the viewer for each pixel. A common type of image-space rendering algorithm is ray tracing [1].

Computer graphics implementations can be further categorized into offline and real-time implementations. The purpose of real-time rendering is to create the illusion of continuity in image rendering and providing interactivity by rapidly rendering each image based on the user's input. This cycle of user input followed by rendering should happen at a fast enough rate to where the viewer is unaware of the fact that each image is rendered in a discrete fashion. The rate at which images are rendered and displayed is measured in frames per second (fps) or Hertz (Hz). Offline rendering implementations on the other hand are not necessarily concerned with the time it takes to render an image [2].

2.2 Ray Tracing
Ray tracing is an image-space based computer graphics technique first described by Whitted [3]. Ray tracing produces images by generating mathematical rays that are projected from the virtual camera in the scene. The rays are tested for intersection with each of the objects that constitute the scene in order to determine visibility and the amount of light that is reflected back along the rays towards the viewer.

The rays initially generated from the camera are called the primary rays. Secondary rays can be generated recursively from the points of intersection with objects in the scene of the primary rays in order to produce additional effects such as shadows, reflection and refraction. Rays can be successively generated from points of intersection of prior rays in order to increase realism or produce additional effects such as reflectivity. The recursive generation can be halted based on some specified depth limit or when the rays stop producing significant information. The ray tracing algorithm is suitable for rendering photo-realistic images since the rays simulate the behavior of light.
Areas and objects under shadows are determined by generating a ray directed towards a light source from an initial point of intersection and testing for objects that are between the initial point of intersection and the light source. If an object is detected by the shadow ray, the initial point of intersection is colored in such a way that it appears to be shaded. Reflection is determined by generating a reflected ray from a previous point of intersection on a reflective object and determining if there is another object in the path of the new reflection ray. If another object is detected, the color of the point of intersection of the original reflective object has the color of the object detected by the reflection ray added to it. [1]

A ray is represented mathematically by a point \( o \) determining the ray’s origin, a vector \( d \) representing the ray’s direction and a parameter \( t \) where \( t = 0 \) at the ray’s origin. Any point on the ray’s trajectory can be expressed as \( p = o + td \).

### 2.2.1 Illumination

Once the visibility of an object in the scene has been determined, a method is needed to determine what color the corresponding pixel should have. The simplest way to color an object is to simply assign some desired color to that pixel once a successful intersection test has been conducted. In order to give the appearance of being three-dimensional on a two-dimensional screen, an illumination model simulating the way light interacts with physical objects should be implemented. A common illumination model that simulates light was first described by Phong [4]. For each material in the scene, the specular, diffuse, ambient, and shininess constants, denoted \( K_s, K_d, K_a, \alpha \) respectively, are defined. Additionally, we have \( L \) which is the direction from the point of intersection to the light point in the scene, the normal \( N \) from the surface at the point of intersection, \( R \) which is a vector representing the direction that a reflected ray originating from the light source would take if it hit the object at the point of intersection and \( V \) which is a vector pointing towards the camera from the point of intersection. The final color \( I \) then becomes
\[ I = K_a I_a + K_d I_d \max(N \cdot L, 0) + K_s I_s \max((V \cdot R)^a, 0) \tag{4.51} \]

Other lighting models for interesting effects and a more physically accurate modification of the Phong model exist [5], however since the focus of this paper is on the intersection aspect of ray tracing, this basic illumination model is presented for a cursory background on the subject of illumination.

### 2.3 Bounding Volume Hierarchy

A bounding volume hierarchy (BVH) is a set of volumes that bound each object in a scene and are arranged in a traversable tree structure. The root of the tree is a bounding volume that entirely encapsulates all objects in the scene with each successive node in the tree encapsulating a subset of smaller bounding volumes all contained within their parent node’s bounding volume. The leaf nodes of the tree finally bound single objects in the scene. Single objects can also be divided into a set of leaf nodes if their computation is expensive. A polygonal mesh may for example be divided into a bounding volume hierarchy so that each node contains a subset of the faces that constitute the mesh. With ray tracing the hierarchy is traversed by testing if the ray intersects each of the child volumes of each node and continuing down the tree from the node which the ray intersects. If the ray intersects several of the child nodes, then the one that is closest is traversed first. This way objects in the scene can be determined as not intersecting the ray since the ray does not intersect their bounding volume. Since ray tracing the objects directly is generally more expensive than ray tracing the bounding volume this tends to speed up the computation of each ray.

Various kind of bounding volumes can be used to construct such hierarchies. Spheres, bounding boxes and axis-aligned bounding boxes being the most common. The type of volume most suitable for the hierarchy usually depends on the types of objects contained within the leaf nodes [6].

Other types of accelerated spatial structures exist such as uniform grids, bounding interval hierarchies, octrees and kd-trees. In [7] a comparative study of all of these types was conducted. The authors concluded that kd-trees are generally faster than bounding volume hierarchies but that bounding volume hierarchies have the advantage of lower memory consumption and a better availability of robust and fast building algorithms. Both kd-trees and bounding volumes hierarchies were faster than the other alternatives.

### 2.4 Embree

Embree is a CPU-based ray tracing API consisting of low-level kernels intended to maximize utilization of modern x86 CPU architectures using Single instruction, multiple data (SIMD) computation for ray traced rendering implementations. Embree allows for runtime code selection that optimizes the accelerated data structures, all of which are variant of bounding volume hierarchies, and build algorithms used in a way that best suits the hardware capabilities and architecture that the ray tracing implementation is running on. Both packet and single ray tracing intersection computations are supported with the ability to dynamically switch between them. Embree has built-in intersection methods for various primitives but also allows for user-defined intersection testing with arbitrary geometries [8].
Embree supports ray packets of sizes 4, 8 or 16 for both determining intersection and occlusion. Embree also uses SIMD math library containing type definition for floating-point variables, double-precision variables, integers and booleans among others that can contain between 4 and 16 separate values in a SIMD variable which allows for computing up to 16 simultaneous computations on each of the values in the variables. Besides these basic types the library also includes second, third and fourth dimensional vectors that can contain the respective variables in each of the elements allowing for easily implemented SIMD vector algebra computation. The SIMD library can be used for computing intersections with user defined geometries by including the library from the Embree source code, available at [9]. The following table described the types used in this thesis.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vfloat8</td>
<td>Basic types that contain eight floats, integers and booleans respectively. The</td>
</tr>
<tr>
<td></td>
<td>Embree source includes overloads and functions for basic arithmetic operations</td>
</tr>
<tr>
<td></td>
<td>as well as fused multiply add (FMA) operations, all using Advanced Vector</td>
</tr>
<tr>
<td></td>
<td>Extensions (AVX) instrinsics</td>
</tr>
<tr>
<td>vint8</td>
<td></td>
</tr>
<tr>
<td>vbool8</td>
<td></td>
</tr>
<tr>
<td>Vec2</td>
<td>Vector types that can contain any of the basic types as elements. Includes</td>
</tr>
<tr>
<td></td>
<td>overloads for arithmetic operations as well as vector specific operations</td>
</tr>
<tr>
<td></td>
<td>such as dot and cross product calculation using AVX instrinsics.</td>
</tr>
<tr>
<td>Vec3</td>
<td></td>
</tr>
<tr>
<td>Vec4</td>
<td></td>
</tr>
</tbody>
</table>

Embree uses a device concept, where an RTCDevice object is instantiated and used for each ray tracing system implemented in the code. These devices don't interfere with each other so that several devices can be used for separate purposes. Embree also provides an RTCScene object which is a container for a list of geometries of potentially various types. Scenes can be created with different options depending on whether ray packets are supported or if the scene is to be dynamic or static.

For a dynamic scene, geometries that are included are enabled by default and can be disabled, re-enabled or removed. Once a geometry has been included in the scene and enabled a call to the API to commit the scene will cause a bounding volume hierarchy to be created for the objects in that particular scene. Objects can later be modified after which a subsequent call to commit the scene will cause the bounding volume hierarchy to be created again. On the other hand, a static scene cannot be modified after the initial call to commit the scene.

Each object that is placed in a scene is given a unique identification number called geomID by Embree. This number can be used to retrieve, modify or delete the geometry once it’s in the scene.
The method with which Embree creates the bounding volume hierarchy for polygonal meshes can be altered by specifying one of the following flags when creating a scene.

Table 2 Flags used to set the conditions for bounding volume hierarchy generation in Embree. The flags are considered suggestions by Embree and may be ignored

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTC_SCENE_COMPACT</td>
<td>Creates a compact data structure and avoids algorithms that consume much memory.</td>
</tr>
<tr>
<td>RTC_SCENE_COHERENT</td>
<td>Optimize for coherent rays (e.g. primary rays).</td>
</tr>
<tr>
<td>RTC_SCENE_INCOHERENT</td>
<td>Optimize for in-coherent rays (e.g. diffuse reflection rays).</td>
</tr>
<tr>
<td>RTC_SCENE_HIGH_QUALITY</td>
<td>Build higher quality spatial data structures.</td>
</tr>
</tbody>
</table>

Embree also provides its own ray structures for various packet sizes that can be passed to a scene in order to check for intersection. The structure for a single ray is given in table 3.

Table 3 the ray object provided by Embree with a description of the associated types.

<table>
<thead>
<tr>
<th>RTCRay</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>org</td>
<td>Ray origin</td>
</tr>
<tr>
<td>dir</td>
<td>Ray direction</td>
</tr>
<tr>
<td>tnear</td>
<td>Start of ray</td>
</tr>
<tr>
<td>tfar</td>
<td>End of ray</td>
</tr>
<tr>
<td>time</td>
<td>Used for motion blur</td>
</tr>
<tr>
<td>mask</td>
<td>Used to mask out geometries</td>
</tr>
<tr>
<td>Ng</td>
<td>Object normal</td>
</tr>
<tr>
<td>u, v</td>
<td>Barycentric coordinates of hit</td>
</tr>
<tr>
<td>geomID</td>
<td>The ID of the hit geometry</td>
</tr>
<tr>
<td>primID</td>
<td>The ID of the primitive hit</td>
</tr>
<tr>
<td>instID</td>
<td>Instance ID of hit</td>
</tr>
</tbody>
</table>

For ray packets the data elements of the ray are eight length arrays of their single ray counterparts. The org, dir and Ng variables are split into three separated eight length arrays for each dimension for the ray packet structure.

The rays are passed to the scene by calling the function rtcIntersect for a single ray or rtcIntersect8 for a packet of eight rays along with the scene that the rays are going to be tested for intersection in. When calling an rtcIntersect function for a ray packet a valid mask also needs to be created and passed to the function. The valid mask is an integer array with each of the values initially set to negative one. When one of the rays in the ray packet intersects an object, Embree sets the corresponding element in the
valid mask to zero. When the packet is later tested against another object, any ray which has its valid mask set to zero is ignored. This also needs to be accounted for manually when implementing user geometries.

A list of geometries supported by Embree can be found at [10]. Embree uses the Möller-Trumbore method [11] for finding intersections between triangles and rays.

User geometries are created in Embree by providing a user created intersection and occlusion function. The functions are passed to Embree using the rtcNewUserGeometry function which also accepts a pointer to the user data that is needed for performing the intersection tests. The user function receives the ray or ray packet and valid mask along with the user geometry and uses this information to fill the ray with data. The data that is passed to the ray is the geomID of the object that has been intersected, setting the tfar member of the ray to the intersection point and setting the normal values of the ray.

2.5 Non-Polygonal Ray Tracing

The popularity of rasterization techniques has caused polygonal meshes to become the geometry representation of choice. So much so that graphics hardware is often designed specifically for the efficient rendering of triangles. Non-polygonal geometry representations generally have to be converted into polygonal meshes by tessellation techniques before rendering with rasterization software and hardware. Efficient tessellation techniques exist for NURBS and Bézier surfaces [12], as well as implicitly defined surfaces [13]. However, tessellation methods can cause artefacts to appear, increase memory usage and generally decrease the scalability and quality of the object that is approximated. Ray tracing on the other hand provides a method for the direct rendering of objects without the need for tessellation.

2.5.1 Ray Tracing Implicit Surfaces

Many methods exist for determining intersection with various forms of implicit surfaces. A comprehensive overview of various methods is given by Hart in [14].

Algebraic surfaces such as ellipsoids and cylinders can be formed into a polynomial function when combined with the equation of a ray. Surfaces such as spheres which form quadratic equations when represented as polynomial functions can be solved using the quadratic equation. Symbolic methods can also be used for finding roots of third and fourth degree polynomials however above that numerical methods need to be applied. An early attempt, mentioned by Hart, using Descartes rule of signs was implemented in [15] Interval analysis which is a numerical method for finding roots within error bounds was used by Mitchell [16]. LG-surfaces proposed by Karla and Barr [17] uses the Lipschitz constants of the implicit function that guarantees finding the roots of the polynomial function where L gives the means to find the regions where a function f is guaranteed to not intersect the surface and G gives the directional derivative along the ray. Sphere tracing described by Hart in [18] marches along the ray towards the implicit surface in steps small enough that they are guaranteed not to miss the surface.

More recently real-time results were achieved using a marching algorithm by Singh and Narayanan. [19]. The method used can render algebraic implicit surface up to an order of 50 as well as non-algebraic surfaces at real-time rates on a GPU. Knoll et al. [20]
presented an algorithm based on interval arithmetic and affine transformation that reached interactive frame rates on both a CPU and GPU implementations using SIMD computation.

2.5.2 Ray Tracing Parametric Surfaces

Methods for determining ray intersections with free-form parametric surfaces such as Bézier surfaces and NURBS can be generally classified as either being based on iterative Newtonian methods or ones based on subdivision.

Initial methods for determining ray intersections with parametric surfaces were first presented by Kajiya [21] and Toth [22] Both employed different techniques based on iterative Newtonian methods in order to find the roots of the polynomial equations, with the former solving the problem of finding the initial values using direct algebraic methods and the latter using interval analysis. In [23] Sweeney and Bartels presented a method for b-spline surfaces that preprocessed the surface into a structure of nested bounding boxes in order to find initial estimates. Martin et al. [24] presented a similar, faster iterative method, also using bounding boxes for initial estimates that focused on simplifying previous methods and presenting the method in a way more suitable for implementation. A further optimization is presented in [25] where only the derivatives of the rational function are calculated instead of both the non-rational and rational derivatives as in Martin’s method.

As for subdivision methods for ray tracing parametric surfaces; a method for intersections with rational Bézier surfaces based on a technique called Bézier clipping was first introduced by Nishita, Sederbergt and Kakimoto in [26]. The technique takes advantage of the convex hull properties of Bézier and NURBS surfaces in order to iteratively remove sections that have not been intersected by the ray. Campagna, Slusallek and Seidel in [27] presented an improved method based on Nishita’s algorithm. The method however requires NURBS surfaces to first be converted into simpler Bézier surfaces in order to work. In [28] the authors detail a problem with the original Bézier clipping method where multiple equivalent intersections can occur causing redundant computation and propose a method for reducing the probability of the phenomenon occurring.

Wang, Zen-Chung and Chuan [29] presented a method based on a combination of Newtonian methods and Bézier clipping that uses ray coherence to find initial values for the Newtonian method. If that were to fail, then the algorithm instead adopts the Bézier clipping method as a substitution.

In [30] Abert, Geimer and Mulle take the recursive basis function for NURBS surfaces and converts it into a non-recursive form optimized for SIMD computation and uses the Newtonian method to find the roots. Several invariants in the surface evaluation are also pre-calculated and stored in a cache. The method was able to achieve greater than one frames per-second results on commodity hardware from 2006. Abert and Geimer also developed a SIMD based implementation for Bézier surfaces in 2005 [31] which achieved between 3 – 5 frames per-second for various models. In a master’s thesis from 2010 [32] another implementation based on Newtonian methods with bounding-volumes for initial value finding running on CUDA compatible GPUs achieved between 3-5 fps for various models, setups and techniques.
In [33] a third alternative for ray tracing parametric surfaces is presented. The method uses the second order derivative of the surface in order to find the roots of the surface equation with the ray. The method is able to better handle special cases than Newton’s method, such as the case where the ray intersects the surface at two points, however according the authors it takes longer to converge than Newton’s method.

3. Method
Initial research in this paper focuses on the various existing methods for representing objects and geometries without the usage of polygonal meshes in computer graphics. Specifically, the non-polygonal representations researched and implemented are ones for which there exist methods to determine ray-object intersections but are traditionally tessellated before rendering with rasterization. The geometry representations chosen for implementation are also constrained by feasibility of completion within the given time-frame of the thesis and the ability to implement with the tools used. The current or future practical usage in real-world implementations is also a discriminating factor.

Methods for measuring common performance related quantities such as frames per second and relative scalability are developed in tandem with the intersection implementations so that performance can be measured and compared. Literature on the subject is also evaluated in order to find other types of measurements that are considered valuable given the context.

Embree is used for developing the respective implementations. The Embree API is used in order to remove the need for implementing custom accelerated structures and hardware parallelism. Embree has built in methods for ray tracing various geometry representations including polygonal meshes but also allows for user defined geometry representations that can use the accelerated structures and SIMD capabilities making it ideal for this work. Using an open source API rather than creating custom implementations also provides an openly available and known point of comparison.

4. Theory
4.1 Implicit Surfaces
An implicit surface is a mathematically defined surface construct defined in three dimensions by an equation \( f(x, y, z) = 0 \). In other words, an implicit surface is defined by the set of points \((x, y, z)\) that satisfy the condition \( f(x, y, z) = 0 \).

Implicit surfaces can be used to represent a wide variety of surfaces. For example a plane as an implicit surface can be defined as \( x + y + z = 0 \) and a sphere with radius \( r \) can be defined as \( x^2 + y^2 + z^2 - r^2 = 0 \). Other examples include ellipsoids, cones, tori and hyperboloids.
Implicit surface geometries can be bounded, i.e. occupying a finite amount of space such as in the case of a sphere, or unbounded such as the case of the plane where the surface extends into infinite space. The degree if an implicit surface is determined by the highest order variable in the defining polynomial, e.g. a sphere is a second degree or quadratic implicit surface since the highest degree for a single variable is two, and a torus is a fourth degree or quartic implicit surface since the highest degree for a single variable is four. Generally it is only possible to analytically solve implicit surfaces up to the fourth degree, beyond that alternative methods need to be applied \[2\].

### 4.2 Parametric Surfaces

A parametric equation assigns a parameter \( t \) to each of the variables in an equation. As an example the equation of a circle with radius \( r = 1 \) defined by \( x^2 + y^2 = 1 \) can parametrically be defined as

\[
S(t) = \begin{cases} 
  x(t) = \cos(t) \\
  y(t) = \sin(t)
\end{cases}
\]

Inserting a value for \( t \) within the defined parametric interval \( 0 \leq t \leq 1 \) will produce a point \((x, y)\) on the circle.

When dealing with three-dimensional parametric equations a second parameter value is introduces. The parametric equation for the sphere with radius \( r \), otherwise defined by

\[
x^2 + y^2 + z^2 = r
\]

is defined parametrically as

\[
S(u, v) = \begin{cases} 
  x(u, v) = r \cdot \sin(u) \cdot \cos(v) \\
  y(u, v) = r \cdot \cos(u) \cdot \cos(v) \\
  z(u, v) = r \cdot \sin(v)
\end{cases}
\]

A set of two parameter values \((u, v)\) would then produce a point \((x, y, z)\) on the sphere. In short, parametric surfaces are defined by taking the function of the surface and deriving a form of the function where each of the variables is defined by a couple of parameter variables \((u, v)\) which for any value within the defined interval produce a point on the surface.
\[ S(u, v) = \begin{cases} x(u, v) \\ y(u, v) \\ z(u, v) \end{cases} \quad u \in [u_{\text{min}}, u_{\text{max}}], v \in [v_{\text{min}}, v_{\text{max}}] \]

### 4.3 Continuity

Continuity describes the differentiability or “smoothness” of a set of curves or surfaces. Continuity is written as \( C^n \) where \( n \) is the amount of times that a set of curves are differentiable at a point. Some examples of levels of continuity including their geometric interpretation are:

- \( C^{-1} \): The curves are discontinuous, i.e. the curves don’t connect to each other.
- \( C^0 \): The curves connect but the tangents differ at the points where they connect.
- \( C^1 \): There is a difference in acceleration at the points where the curves connect.
- \( C^2 \): The point at which the curves connect have the same acceleration.

### 4.4 Bézier Curves and Surfaces

#### 4.4.1 Bézier Curves

A Bézier curve is a polynomial curve that proportionally shapes itself relative to a set of control points. The curves can be a polynomial of any degree and a curve of degree \( n \) will be controlled by \( n + 1 \) control points. For computer graphics implementations it is common to approximate a shape using a set of connected cubic Bézier curves; such linear combinations of polynomials are called splines. The curve does not typically intersect the individual control points of the curve, except for the beginning and ending control points, rather the curve is contained within the *convex hull* formed by the control points. The shape of the curve can be directly altered by manipulating the position of the individual control points making them ideal for computer aided design and modelling.

A Bézier curve is written as

\[
C(t) = \sum_{i=0}^{n} P_i B_{i,n}(t) 
\]

\[ 0 \leq t \leq 1 \]  

Where \( n \) is the degree of the curve, \( P_i \) are a set of two or three-dimensional control points, where the amount of control points is one more than the degree of the curve that they shape. \( B_{i,n} \) are the blending functions of the curve defined as
\[ B(t)_{i,n} = \frac{n!}{n!(n-i)!} t^i (1-t)^{n-i} \quad i = 0, ..., n \] (4.6)

As an example, given two control points, \( P_0 \) and \( P_1 \), defining a linear or first degree curve, equation 4.5 expands to

\[ C(t) = (1-t)P_0 + tP_1 \]

which is a finite straight line starting at \( P_0 \) and ending at \( P_1 \) given that \( 0 \leq t \leq 1 \).

If the control points are \( P_0, P_1, P_2, P_3 \) then the curve is cubic and is given by

\[ C(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3 \quad 0 \leq t \leq 1 \] (4.7)

with the terms

\[
\begin{align*}
(1-t)^3 & \\
3(1-t)^2 & \times t \\
3(1-t) & \times t^2 \\
t^2 & 
\end{align*}
\] (4.8)

given by the blending functions for Bézier curves.

As can be seen a change in any of the control points results in a change of the entire curve; single Bézier curves are thus limited in the shapes that they can represent which means that a larger set of connected curves is required to represent closed shapes such as ellipsoids. However, limits also exist in the extent to which Bézier curves can be connected.

### 4.4.2 Continuity between Bézier Curves

Given two cubic curves with control points \( P_0, P_1, P_2, P_3 \) and \( Q_0, Q_1, Q_2, Q_3 \), simply setting the control points such that \( P_3 = Q_0 \) would create a composite curve or spline.

However merely placing the beginning and ending control points in the same place produces only \( C^0 \) continuity. In order to achieve \( C^1 \) continuity, the following rule must be applied

\[ m(P_m - P_{m-1}) = n(Q_1 - Q_0) \] (4.9)

Equation 4.9 entails that in order to achieve \( C^1 \) continuity between two connected Bézier curves the ratio of the distance between the last two control points of the first curve and the distance between the two control points of the second curve must be \( n/m \). Where \( m \) is the degree of the first curve and \( n \) is the degree of the second. Achieving \( C^2 \) continuity or above is not possible for Bézier curves, although control points can always be positioned such that they at least appear to be continuous to a larger extent.
### 4.4.3 De Casteljau’s Algorithm

Finding a point on a Bézier curve given a parameter value \( t \) can be achieved by simply inserting \( t \) and doing the calculations as per the definition of Bézier curves. The problem with this method in terms of computation is that it is not very numerically stable. A more stable approach is given by De Casteljau’s algorithm [34].

The algorithm recursively linearly interpolates a point between each set of two adjacent control points until the desired value is found. Linearly interpolating a point \( t \in [0,1] \) between two points \( P_0 \) and \( P_1 \) is done by

\[
P(t) = P_0(1 - t) + P_1 t
\]  

(4.10)

The point \( P(t) \) is then the point at parameter value \( t \) along the line \( P_0 \) to \( P_1 \). The way that De Casteljau’s Algorithm works is that given a set of points \( P_0, P_1 \ldots P_n \), linearly interpolate a new point at \( t \) along each successive pair of points \( P_0 \) to \( P_1 \ldots P_{n-1} \) to \( P_n \). This gives a new set of \( n \) points. Continue the algorithm with the new set of points which gives another set of \( n - 1 \) points. If done \( n \) times, where \( n \) is the degree of the curve, the result is a single point. This point is the point \( t \) along the original curve.

The algorithm can be expressed with the following recursive equation

\[
P_{i,j} = (1-t)P_{i-1,j} + uP_{i-1,j+1} \quad \{ i = 0, 1, \ldots, n \}
\]

\[
j = 0, 1, \ldots, n - i
\]

(4.11)

Besides finding a point \( P(t) \) on the curve, the algorithm can also be used to divide the original curve at a point at \( t \). The algorithm is simply applied at the split point \( t \) until two new sets of control points can be retrieved which respectively describe both halves of the original curve.

An important note to make is that even though the convex hull property of the curve is always maintained when subdividing; the new control points will become closer and closer to the curve as higher levels of subdivision are applied, allowing for the creation of a successively tighter convex hull or control net.

### 4.4.4 Bézier Surfaces

A Bézier surface is formed by the product of the blending functions of two orthogonal Bézier curves.

\[
S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i} \frac{m!}{j!(m-j)!} v^j (1-v)^{m-j}
\]

(4.12)

\[
0 \leq u \leq 1 \\
0 \leq v \leq 1
\]

Or more compactly as

\[
S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_{i,n}(u) B_{j,m}(v)
\]

(4.13)
Like Bézier curves, Bézier surfaces are shaped by manipulating the associated control points and don't generally intersect with the individual control points except for those situated on the corners of the surface. The surfaces are also contained within the convex hull formed by the control points. The same rules regarding continuity also apply. De Casteljau’s algorithm can be applied to a Bézier surface by applying it to each set of control points in both u and v direction.

![A Bicubic Bézier Surface with 16 control points](image)

**4.4.5 Partial Derivatives**

The derivatives of a Bezier surface have many uses such as finding the normal at a point or evaluating curvature. The partial derivatives in u and v direction are

\[
B_u(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_i^{'}(u) B_j^{'}(v) \tag{4.14}
\]

\[
B_v(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{i,j} B_i^{'}(u) B_j^{'}(v) \tag{4.15}
\]

The normal at the point \((u, v)\) can be found by normalizing the cross product of the two partial derivatives.

**4.4.6 Matrix Representation**

It is useful for many applications to represent a Bezier surface in matrix form. The way in which a cubic Bézier surface is represented as a matrix is described in [35] and will be outlined here.

A Bezier surface can be represented by the matrices

\[
B(u, v) = [U][N][CP][N]^T[V]^T \tag{4.16}
\]

where for an \(n^{th}\) degree surface
\[ [U] = [u^n \ u^{n-1} \ ... \ 1] \]
\[ [V] = [v^n \ v^{n-1} \ ... \ 1] \]  
\[ (4.17) \]

\[ [N] \] is an \((n + 1)\) by \((n + 1)\) matrix containing the coefficients for the Bezier blending functions which for a cubic Bezier surface has the form

\[ [N] = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \]  
\[ (4.18) \]

The matrix \([CP]\) contains the respective control points in both directions, making \([CP]\) a \((n + 1)\) by \((n + 1)\) matrix of three-dimensional vectors.

\[ [CP] = \begin{bmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{bmatrix} \]  
\[ (4.19) \]

The partial derivatives in both \(u\) and \(v\) direction can easily be represented in matrix form as

\[ B_u(u,v) = [U'][N][CP][N]^T[V]^T \]
\[ B_v(u,v) = [U][N][CP][N]^T[V']^T \]  
\[ (4.20) \]

where

\[ [U'] = [u^{n-1} \ u^{n-2} \ ... \ 1 \ 0] \]
\[ [V'] = [v^{n-1} \ v^{n-2} \ ... \ 1 \ 0] \]  
\[ (4.21) \]

### 4.5 Non-Uniform Rational B-Spline (NURBS) Curves and Surfaces

#### 4.5.1 NURBS Curve

Considering that non-uniform rational b-splines (NURBS) are the most general cases of b-splines it is beneficial to begin with the definition of b-splines and build towards the definition of NURBS more generally. B-spline or basis-spline curves are similar to Bezier curves in that they shape themselves relative to a set of control points. B-spline curves however offer increased flexibility over Bézier curves by allowing for piecewise construction of several polynomial curves in a single spline construction. In other words, b-spline curves are not constrained by the \(n + 1\) control points rule for a \(n^{th}\) degree curve, but rather can increase the set of control points in order to increase flexibility and control as long as the number of control points are equal to or greater than the degree of the curve plus one. This increase in flexibility however entails an increase in computational and structural complexity.
Besides the shape being determined by the set of control points and the degree, b-splines have an associated knot vector which defines the parameter intervals in which the control points provide local support and at which points the splines of a curve connect. Mathematically a b-spline curve is defined similarly to a Bezier curve by the polynomial

\[ f(t) = \sum_{i=0}^{n} P_i B_i(t) \]  

(4.22)

Where \( P_i \) are the control points, \( n \) is the number of control points and \( B_i(t) \) is the basis-function with respect to the parameter \( t \).

Each of the individual control points can also have an associated weight \( w_i \). The weights can be increase in order for the associated control point to have an increased effect on the curve or decreased in order to reduce the effect on the curve. A b-spline with weights is defined by the polynomial

\[ f(t) = \frac{\sum_{i=0}^{n} w_i P_i B_i(t)}{\sum_{i=0}^{n} w_i B_i(t)} \]  

(4.23)

A b-spline curve defined in such a way is called a rational curve given the rational form of the defining polynomial. In the case where all weights are equal to one, equation 4.23 reduces to equation 4.22, making the non-rational case a special case of the rational one. The weights of a b-spline curve must never be negative since that could break the convex hull property of the curve. By multiplying the control points by the weights and appending the weights to the ends of each control point a more compact form of equation 4.23 is

\[ f(t) = \sum_{i=0}^{n} p^{h_i} B_i(t) \]  

(4.24)

Where \( p^{h_i} \) denotes that the control points have had their weights appended and multiplied.

4.5.2 Basis Functions
The basis-functions that smoothly blend the curve given the set control points and parameter are given by the recursive Cox De-Boor formula defined by

\[ B_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases} \]  

(4.25)
Here $p$ is the degree of the curve and $i$ is given by equation 4.23. As can be seen the basis-functions are evaluated recursively starting from degree $p$ down to the $0^{th}$ degree, meaning that for an increased degree the depth of the recursive formula increases and for an increase in control points the amount of basis-functions evaluated increases. The convention $\frac{0}{0} = 0$ is used. $u_i$ is the $i^{th}$ element of the curve’s knot vector.

### 4.5.3 Knot Vector

The knot vector of a b-spline curve is defined by $n + p + 1$ elements where $n$ is the number of control points and $p$ is the degree. Mathematically the knot vector can be written as

$$\vec{u} = [u_0 \ u_1 \ ... \ u_{n+p}] \tag{4.27}$$

Each of the elements in the knot vector determine the points in the parameter domain at which the splines of the b-spline curve connect to each other or ‘tie-together’. The minimum and maximum elements in a knot vector can be any number, including negative ones, i.e. the magnitude of the vector does not matter. However, it is common to normalize the elements such that they are between zero and one. The elements in the knot vector must be set in a non-decreasing order, that is, $u_i \leq u_{i+1}$ for all elements in the knot vector meaning that $[1 \ 2 \ 3 \ 4 \ 5 \ 6]$ is a valid knot vector whereas $[6 \ 5 \ 4 \ 3 \ 2 \ 1]$ is not.

Each successive pair of elements in the knot vector $u_i ... u_{i+1}$ are referred to as a knot span and represent a particular parameter interval. The only valid knot spans are within the parameter domain $u_{\text{min}} \leq u < u_{\text{max}}$ with $u_{\text{min}} = u_p$ where $p$ is the degree of the curve and $u_{\text{max}} = u_n$ where $n$ is the number of control points. Any parameter value outside of this span is not defined for the curve.

A non-empty knot span is any span where $u_i \neq u_{i+1}$. Such spans define a single polynomial within the spline and therefore have $C^p$ continuity within that span, where $p$ is the degree of the curve. Increasing the number of equal knots in a knot span to some number produces an empty knot span and increases that span’s multiplicity. Increasing multiplicity decreases the differentiability or continuity of the basis-functions defined by the span to $C^{p-k}$ where $p$ is the degree and $k$ is the multiplicity. As an example, take a cubic curve with four control points and multiplicity $k = 4$

$$\vec{u} = [0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1] \tag{4.28}$$

The valid knot span of this curve is $u_3 \leq u < u_4$. So $u$ is defined between 0 and 1 with a single polynomial curve of $C^p$ continuity. In fact, if setting all of the weights to one for this particular curve, it is geometrically equivalent to that of a Bézier curve.
If we instead take a cubic curve with five control points and the knot vector

$$\bar{u} = \begin{bmatrix} k=4 & k=1 & k=4 \\ 0 & 0 & 0 & 0.5 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(4.29)

Then we have a spline defined by two polynomial curves the first of which is defined within the parameter \( u_1 \in [0, 0.5] \) and the second \( u_2 \in [0.5, 1] \). The two polynomial curves meet at the point \( u = 0.5 \) and have \( C^{p-k} = C^2 \) continuity at that point. Increasing the multiplicity of the middle knot element would decrease the continuity at that point but would also necessitate increasing either the degree (which would consequently make the continuity unaltered) or increasing the number of control points. Increasing the degree may change the shape of the curve so it is necessary to increase the amount of control points if maintaining the original form is required. The process of inserting new knots into a knot vector while maintaining the same original shape is called knot insertion, described in more detail in section 4.4.5

Knot vectors can be classified into three different types: uniform, open uniform and non-uniform.

Uniform knot vectors are knot vectors for which the spacing between each of the elements is equal. An example would be \([0 1 2 3 4 5 6]\) or \([0.0 0.1 0.2 0.3 0.4 0.5 0.6]\). Uniform knot vectors can be used to describe closed shapes by making the endpoints coincide.

An open uniform knot vector has equidistant knots except for the beginning and ending knots which have multiplicity \( p + 1 \) where \( p \) is the degree of the curve. An example of an open uniform curve would be \([0 0 0 0 0.25 0.5 0.75 1 1 1 1]\) for a cubic spline with seven control points.

Lastly, non-uniform knot vectors are the most general form of knot vectors where the only constraint is that the elements of the knot vector are in ascending order. Both uniform and open uniform knot vectors are special cases of non-uniform knot vectors.

4.5.4 Local Support

It follows from equation 4.25 and 4.26 that any basis function \( B_{i,p}(t) \) is non-zero only within the knot span \([t_i, ..., t_{i+p+1}]\). This property is referred to as local support which leads to the property of local modification which says that a control point \( P_i \) only has an effect on the curve within the interval \([t_i, ..., t_{i+p+1}]\), which allows the curve to be modified within a certain parameter interval without the rest of the curve being affected. The parameter intervals in which specific control points or sets of control points have an effect on the curve is determined by the knot vector.

4.5.5 Knot Insertion

It is possible to insert new knots into a knot vector while maintaining the original shape and increasing the amount of control points. Knot insertion is not only useful for allowing a higher degree of fine tuning, but also finds use in subdivision algorithms.

Given a b-spline curve defined by
\[ f(t) = \sum_{i=1}^{n} P_i B_i(t) \] (4.30)

and knot vector

\[ \bar{u} = [u_0 \ u_1 \ ... \ u_l \ u_{l+1} \ ...] \] (4.31)

Inserting a new knot between \( u_l \) and \( u_{l+1} \) gives the new curve

\[ f(t) = \sum_{i=1}^{n+1} Q_i B_i(t) \] (4.32)

Where the new control points \( Q_i \) are given by

\[ Q_i = (1 - \alpha_i) P_{i-1} + \alpha_i P_i \] (4.33)

\[ \alpha_i = \begin{cases} 1 & i \leq l - k + 1 \\ 0 & i \geq l + 1 \\ \frac{\bar{u} - u_i}{u_{l+k-1} - u_i} & l - k + 2 \leq i \leq l \end{cases} \] (4.34)

where \( \bar{u} \) is the new knot and \( k \) is the order of the curve, i.e. the degree of the curve plus one. Although the new curve has a different knot vector and different set of control points, it is geometrically equivalent to its previous shape. The algorithm for inserting one knot at a time is known as Boehm’s algorithm [36]. A more general form that handles the insertion of multiple knots simultaneously is known as the Oslo algorithm [37], [38] which is briefly detailed here.

Given a knot vector

\[ \bar{u} = [u_0 \ u_1 \ ... \ u_{n+p+1}] \] (4.35)

Inserting an arbitrary amount of new knots will give the new knot vector

\[ \vec{v} = [v_0 \ v_1 \ ... \ v_{m+p+1}] \] (4.36)

Where \( m > n \) and corresponds to the number of inserted knots. The new curve is then defined by
\[ g(t) = \sum_{j=0}^{m} Q_j^h B_j(t) \]  

(4.37)

Since the new curve is geometrically equivalent to the previous one, the new control points can be computed based on the identity \( f(t) = g(t) \) where \( f(t) \) is the old curve defined by the knot vector \( \bar{u} \). With this in mind; the new control points are derived using the equations.

\[ Q_j^h = \sum_{i=0}^{m} P_j^h \alpha_{i,j}^p \]  

(4.38)

\[ \alpha_{i,j}^0 = \begin{cases} 1 & u_i \leq v_j \leq u_{i+1} \\ 0 & \text{otherwise} \end{cases} \]  

(4.39)

\[ \alpha_{i,j}^p = \frac{v_{i+p} - u_i}{u_{i+p} - u_i} \alpha_{i,j}^{p-1} + \frac{u_{i+p-1} - v_{j+p}}{u_{i+p-1} - u_{i+1}} \alpha_{i+1,j}^{p-1} \]  

(4.40)

4.5.6 NURBS Surface

A NURBS surface is created by the tensor product of two NURBS curves

\[ S(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) P_{i,j} w_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) w_{i,j}} \]  

(4.41)

Where \( p, q \) are the respective degrees of the curves. Each curve in the construction has a separate knot vector and can therefore each have a separate number of control points. A more succinct form of the equation where the weights have been appended to and multiplied with the control points is

\[ S(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i(u) B_j(v) P_{i,j}^h \]  

(4.42)

All of the properties associated with NURBS curves, such as the convex hull property and local support, are carried over to the surface form.
4.5.7 Partial Derivatives

The derivatives of a NURBS surface can be used for finding the normal at a point or evaluating curvature among other useful properties. In order to find the derivative of the NURBS surface the numerator and denominator are represented as separate functions as follows:

\[
S(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,p}(u) B_{j,q}(v) P_{i,j} w_{i,j}}{\sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,p}(u) B_{j,q}(v) w_{i,j}} = \frac{N(u, v)}{D(u, v)} \quad (4.43)
\]

The quotient rule can then be applied which produces

\[
S_u(u, v) = \frac{D(u, v) N_u'(u, v) - N(u, v) D_u'(u, v)}{[D(u, v)]^2} \quad (4.44)
\]

\[
S_v(u, v) = \frac{D(u, v) N_v'(u, v) - N(u, v) D_v'(u, v)}{[D(u, v)]^2} \quad (4.45)
\]

Which are the partial derivatives of a NURBS surface in both u and v direction where

\[
N_u'(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B'_{i,p}(u) B_{j,q}(v) P_{i,j} w_{i,j} \quad (4.46)
\]

\[
N_v'(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,p}(u) B'_{j,q}(v) P_{i,j} w_{i,j} \quad (4.47)
\]
\[ D_u'(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,p}(u)B_{j,q}(v)w_{i,j} \] \hfill \text{(4.48)}

\[ D_v'(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_{i,p}(u)B_{j,q}(v)w_{i,j} \] \hfill \text{(4.49)}

and the derivative of the basis function is given by

\[ B_{i,p}'(t) = \frac{p}{t_{i+p} - t_i}B_{i,p-1}(t) - \frac{p}{t_{i+p+1} - t_{i+1}}B_{i+1,p-1}(t) \] \hfill \text{(4.50)}

Note that \( B_{i,p-1}(t) \) and \( B_{i+1,p-1}(t) \) in equation 4.50 refer to the original Cox De-Boor recurrence and not the derivative, i.e. the recursion continues down the original basis function where only the first layer is changed to the equation 4.50.

### 5 Ray Tracing Implicit Surfaces

#### 5.1 Ray-Sphere Intersection

A common method for finding the point of intersection between a ray and a sphere is given in [39]. The sphere is defined by the center \( c \), radius \( r \) and a point \( p \) on the sphere by

\[ (p - c)^2 - r^2 = 0 \] \hfill \text{(5.1)}

To determine a point of intersection the point \( p \) is substituted with the ray equation

\[ (o + td - c)^2 - r^2 = 0 \] \hfill \text{(5.2)}

Expanding equation 5.2 gives

\[ (d^2)t^2 + 2dt(o - c) + (o - c)^2 - r^2 = 0 \] \hfill \text{(5.3)}

Which is a quadratic equation for \( t \) that can be written as

\[ at^2 + bt + c = 0 \]

\[ a = d^2 \]

\[ b = 2d(o - c) \]

\[ c = (o - c)^2 - r^2 \]
the solution for the equation is

\[ t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]  

(5.5)

Quadratic equations have zero, one or two roots which can be determined by the discriminant

\[ b^2 - 4ac \]  

(5.6)

If the value of the discriminant is less than zero then the ray does not intersect the sphere, if the discriminant is equal to zero then the ray grazes the sphere at a single point and if the discriminant is greater than zero then the ray goes through the sphere. Two points of intersection can then be determined from the equation; one where the ray enters the sphere and one where it exits. The point nearest the camera becomes the point used for determining the color of the corresponding pixel on the screen.

### 5.2 Ray-Cylinder Intersection

The equation for an infinite cylinder oriented along the line \( c = o_c + t\epsilon d_c \) with radius \( r \) is given by

\[ (q - o_c - ((d_c \cdot q - o_c) \cdot d_c)^2 = r^2 \]  

(5.7)

where \( q = (x, y, z) \) is a point on the curve. Substituting the point with the ray equation \( q = o + td \), the cylinder equation can be transformed into the quadric form \( at^2 + bt + c = 0 \) where

\[ a = (d - (d \cdot d_c) \cdot d_c)^2 \]
\[ b = 2((d - (d \cdot d_c)) \cdot ((o - o_c) - ((o - o_c) \cdot d_c)d_c) \]
\[ c = ((o - o_c) - ((o - o_c) \cdot d_c)d_c)^2 - r^2 \]  

(5.8)

The solution to the equation is given by

\[ t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]  

(5.9)

where a non-negative discriminant

\[ b^2 - 4ac \geq 0 \]  

(5.10)

entails that the ray intersects the cylinder. In order to have a finite cylinder the ray is tested to see if it between the two planes determined by the line around which the
cylinder revolves by

\[ d_c \cdot (q - o_c) > 0 \]  
\[ d_c \cdot (q - c) < 0 \]  \hspace{1cm} (5.11)

Where \( q \) is the ray equation with \( t \) set to the point of intersection and \( c \) is the line that the cylinder revolves around with \( t_c \) set to the length of the cylinder.

Rendering a closed cylinder requires testing for intersection with the planes that bound the cylinder. The equations for planes contained within the cylinder’s radius \( r \) are given by

\[ d_c \cdot (q - o_c) = 0 \text{ and } (q - o_c)^2 < r^2 \]  
\[ d_c \cdot (q - c) = 0 \text{ and } (q - c)^2 < r^2 \]  \hspace{1cm} (5.12)

Finding the final point of intersection is done by retrieving the non-negative results from equation 5.9 and the non-negative results from equations 5.12 The smallest of these values then determines the point of intersection [40].

6. Ray Tracing Parametric Surfaces

The Newtonian method for finding an intersection with a ray and both NURBS surfaces and Bézier surfaces differs only in the surface evaluation. Therefore, this section is dedicated to detailing the aspects of the intersection method that both types of surfaces have in common. The differences will be detailed in their respective dedicated sections. The Newtonian method for ray tracing parametric surfaces described here is based on the methods described by Martin in [24] with the additional enhancement developed by Abert [30].

6.1 Newton Rhapson

The Newton-Rhapson method, is an iterative method for finding the roots of a function. Given an initial guess, a function and its derivative; the Newton method for a function of a single variable is as follows

\[ x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \]  \hspace{1cm} (6.1)

Where \( x_0 \) is the initial guess and \( x_1 \) is a new value for \( x \) closer to the root given that the iteration converged towards the root.

The more general form of the method for the \( i^{th} \) iteration is
\[ x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \] (6.2)

With a good initial guess the rate of convergence is quadratic. However, a bad guess or local extrema may cause the iteration to diverge from the root.

### 6.2 Ray Representation

The ray for which an intersection with the surface is to be found is normally represented by the equation

\[ r = o + td \] (6.3)

Using a technique developed by Kajiya [21] for the Newtonian method, the ray is instead represented as the intersection of two orthogonal planes each represented as \( p_1 = (n_1, d_1) \) and \( p_2 = (n_2, d_2) \) where \( n_1 \) and \( n_2 \) are two unit length vectors that are orthogonal to each other and \( d_1, d_2 \) represent the distance of each plane to the origin.

The original ray representation can be transformed into the two planes by applying the following

\[ n_1 = \begin{cases} \langle d_y, -d_x, 0 \rangle & \text{if } |d_x| > |d_y| \text{ and } |d_x| > |d_z| \\ \langle 0, d_x - d_y \rangle & \text{otherwise} \end{cases} \] (6.4)

Applying the above rule causes \( n_1 \) to always be perpendicular to the ray direction \( d \). \( n_2 \) can then simply be retrieved by

\[ n_2 = n_1 \times d \] (6.5)

and the respective distances are given by

\[ d_1 = -n_1 \cdot o \]
\[ d_2 = -n_2 \cdot o \] (6.6)

### 6.3 Newton’s Method for Ray-Surface Intersection

Finding the intersection between a ray and a surface turns into the problem of finding the roots for the bivariate equation

\[ R(u, v) = \begin{cases} n_1 \cdot S(u, v) + d_1 \\ n_2 \cdot S(u, v) + d_2 \end{cases} \] (6.7)

Where \( S(u, v) \) is the point given by evaluating either the NURBS surface or the Bézier surface at a point in terms of \( u \) and \( v \). In geometric terms \( |R(u, v)| \) can be interpreted as the distance between the ray and the evaluated surface point \( S(u, v) \). Since the
Newtonian method iteratively approximates the roots, an intersection is reported once the distance comes under a certain user defined threshold

$$|R(u, v)| < \varepsilon$$  \hspace{1cm} (6.8)

The iteration is stopped whenever any new u, v values given from the iteration results in a point further away from the ray

$$|R(u_{i+1}, v_{i+1})| > |R(u_i, v_i)|$$  \hspace{1cm} (6.9)

Lastly no intersection is reported if the number of Newton iterations surpasses some user defined threshold.

Finding the new u, v values given the old ones is done by applying

$$\begin{pmatrix} u_{i+1} \\
v_{i+1} \end{pmatrix} = \begin{pmatrix} u_i \\
v_i \end{pmatrix} - J^{-1}R(u, v)$$ \hspace{1cm} (6.10)

J is known as the Jacobian matrix and is given by

$$J = \begin{pmatrix} n_1 \cdot S_u(u, v) & n_1 \cdot S_v(u, v) \\ n_2 \cdot S_u(u, v) & n_2 \cdot S_v(u, v) \end{pmatrix}$$ \hspace{1cm} (6.11)

and

$$J^{-1} = \frac{1}{\det(J)} \begin{pmatrix} n_2 \cdot S_v(u, v) & -n_1 \cdot S_v(u, v) \\ -n_2 \cdot S_u(u, v) & n_1 \cdot S_u(u, v) \end{pmatrix}$$ \hspace{1cm} (6.12)

$$\det(J) = (n_1 \cdot S_u(u, v)) * (n_2 \cdot S_v(u, v)) - (n_2 \cdot S_u(u, v)) * (n_1 \cdot S_v(u, v))$$ \hspace{1cm} (6.13)

Equation 6.10 is as can been seen analogous to the Newton Rhapson formula (equation 6.2)

A problem, noted in [24], may arise if the determinant $\det(J)$ is singular, i.e. $\det(J) \approx 0$. Such a determinant may make it difficult to determine the inverse Jacobian. The problem can be circumvented by performing a small random perturbation of the u, v values after checking if the Jacobian is near zero and initiating the next iteration if it is.

$$\begin{pmatrix} u_{i+1} \\
v_{i+1} \end{pmatrix} = \begin{pmatrix} u_i \\
v_i \end{pmatrix} + 0.1 \begin{pmatrix} \text{rand}(0,1) * (u_0 - u_i) \\
\text{rand}(0,1) * (v_0 - v_i) \end{pmatrix}$$ \hspace{1cm} (6.14)
Where \(u_0\) and \(v_0\) are the initial guesses and \(\text{rand}(0,1)\) produces a random value between 0 and 1.

Since the final point retrieved is an approximation and most likely does not lie on the ray’s path a projection of the point onto the ray is performed using

\[
t = (S(u, v) - o) \cdot d
\]  

(6.15)

### 6.4 Bézier Surface Evaluation

Evaluating a point on a bicubic Bézier surface given a set of \(u, v\) values becomes an easier task if the surface is represented in matrix form [31] by

\[
S(u, v) = [U][N][CP][N]^{T}[V]^{T}
\]  

(6.16)

As previously mentioned, \([U]\) and \([V]\) are \([u^n \ u^n-1 \ldots 1]\) and \([v^n \ v^{n-1} \ldots 1]\) respectively, \([N]\) contains the polynomial coefficients and \([CP]\) is a 4x4 matrix of three-dimensional vectors.

Since the product \([N][CP][N]^{T}\) only ever changes when the control points change, it can be computed during the preprocessing step and stored in a separate matrix. It’s also worth noting that as long as \([U]\) and \([V]\) are represented in the order described where the elements start from \(u^n\) to 1 then \([N]\) is equal to its transpose \([N]^{T}\) and therefore the transpose need not be calculated.

The surface evaluation begins with the calculation of \([U]\), \([V]\), \([U']\), and \([V']\). After which the products \([U][N][CP][N]^{T}\) and \([U'][N][CP][N]^{T}\) are determined. The calculation then continues for \([U][N][CP][N]^{T}[V]\), \([U'][N][CP][N]^{T}[V]\) and \([U][N][CP][N]^{T}[V']\)

Doing the calculation of both the point on the surface as well as the partial derivatives in the same scope allows for the product \([U][N][CP][N]^{T}\) to be pre-calculated and used for the calculation of \([U][N][CP][N]^{T}[V]\) and \([U][N][CP][N]^{T}[V']\) considering the products that they have in common which decreases the amount of calculations needed compared with doing the surface evaluation for the point and the derivatives separately.

The results of these calculations are three three-dimensional vectors representing the point of the surface corresponding to the given \(u, v\) values and the partial derivatives in \(u\) and \(v\) direction respectively.

### 6.5 Bézier Surface Subdivision

In order to create the bounding volume hierarchy for Bézier surfaces, the surface needs to be successively subdivided until a certain flatness criterion is met. De Casteljau’s Algorithm (described in section 4.3.3) can be used for successively subdividing a Bézier surface at a parameter value \(t\). The algorithm is applied to each of the curves that compose the surface in a single direction. The following shows how to subdivide a cubic Bézier curve at \(t = 0.5\).

Linearly interpolating a point between two points is done by
\[ P(t) = P_0(1 - t) + P_1t \]  
(6.17)

Which for \( t = 0.5 \) simplifies to

\[ P(t) = (P_0 + P_1) \cdot 0.5 \]  
(6.18)

Given the original four control points \( P_0, P_1, P_2, P_3 \) and parameter \( t = 0.5 \). A new set of linearly interpolated points \( Q_0, Q_1, Q_2 \) is given by

\[
\begin{align*}
Q_0 &= (P_0 + P_1) \cdot 0.5 \\
Q_1 &= (P_1 + P_2) \cdot 0.5 \\
Q_2 &= (P_2 + P_3) \cdot 0.5
\end{align*}
\]  
(6.19)

Finally, these points along with the original points are used to construct the final set of \( 2n + 1 \) points \( R_0, R_1, R_2, R_3, R_4, R_5, R_6 \) by

\[
\begin{align*}
R_0 &= P_0 \\
R_1 &= Q_0 \\
R_2 &= (Q_0 + Q_1) \cdot 0.5 \\
R_4 &= (Q_1 + Q_2) \cdot 0.5 \\
R_3 &= (R_2 + R_4) \cdot 0.5 \\
R_5 &= Q_2 \\
R_6 &= P_3
\end{align*}
\]  
(6.20)

Which produces the new pair of curves \( R_0, R_1, R_2, R_3 \) and \( R_3, R_4, R_5, R_6 \). Applying the algorithm to each set of curves in a single direction of the surface will split the surface at \( t = 0.5 \). Splitting the two new surfaces in the other direction will produce four new surfaces.

### 6.5.1 Bézier Flatness Test

Performing subdivision with De Casteljau’s algorithm is both accurate, robust and the amount of times that a subdivision is performed can easily be controlled by stopping the subdivision process once a specified depth has been reached. Doing so might however lead to already flat surfaces being subdivided more than is needed. In order to halt the subdivision process when the surface is reasonably flat, a flatness metric is needed which can be used to stop the subdivision once it has been reached. A robust and accurate method for determining flatness can be found at [41] which will be shortly outlined here.

The method basically measures the distance from the curve at parameter value \( t \), to where a point on the curve at \( t \) would be if the curve were a straight line. Given the control points \( P_0, P_3 \), a straight lined cubic curve is given by

\[ S(t) = (1 - t)^3P_0 + 3(1 - t)^2t \left( \frac{2}{3}P_0 + \frac{1}{3}P_3 \right) + 3(1 + t)t^2 \left( \frac{1}{3}P_0 + \frac{2}{3}P_3 \right) + t^3P_3 \]  
(6.21)
which is simply a cubic curve with control points \( P_0, \frac{2}{3} P_0 + \frac{1}{3} P_3, \frac{1}{3} P_0 + \frac{2}{3} P_3, P_3 \) that describe a straight line between \( P_0, P_3 \) as a function of \( t \). Given the cubic curve \( B(t) \), the flatness of which is to be evaluated, the distance between the two curves at \( t \) is given by

\[
d(t) = \|B(t) - S(t)\|
\]  \(\text{(6.22)}\)

Simplifying algebraically, the square of this expression can be shown to be

\[
d^2(t) = (1 - t)^2 t^2 \|a + tb\|^2
\]  \(\text{(6.23)}\)

\[
a = 3P_1 - 2P_0 - P_3
\]

\[
b = 3P_2 - P_0 - 2P_3
\]  \(\text{(6.24)}\)

The maximum value of \((1 - t)^2 t^2\) can be shown to be \(1/16\) within the interval \(0 \leq t \leq 1\), and if \(a = (a_x, a_y, a_z)\), \(b = (b_x, b_y, b_z)\), then

\[
\|(1 - t)a + tb\|^2 = ((1 - t)a_x + tb_x)^2 + ((1 - t)a_y + tb_y)^2 + ((1 - t)a_z + tb_z)^2
\]  \(\text{(6.25)}\)

The maximum of each term is given by \(\max(a_x^2, b_x^2), \max(a_y^2, b_y^2)\) and \(\max(a_z^2, b_z^2)\), which finally gives an expression for the maximum distance of the distance function \(d^2(t)\) within the parameter bounds \(0 \leq t \leq 1\) as

\[
\frac{1}{16} (\max(a_x^2, b_x^2) + \max(a_y^2, b_y^2) + \max(a_z^2, b_z^2))
\]  \(\text{(6.26)}\)

The expression can be used to stop subdividing a Bézier curve once the value of it is below some user defined threshold.

### 6.6 NURBS Surface Evaluation

When evaluating a point and the derivatives of a NURBS surface given a set of \(u, v\) values the local support property should be taken into account. A naïve approach would compute all of the basis-functions for each iteration, including the ones which are zero, whereas accounting for the fact that each parameter value of \(u\) and \(v\) lies in exactly one valid knot span, only \(p + 1\) higher order basis functions are non-zero. A method for evaluating only the non-zero basis functions is given in [32] and outlined here.

When evaluating only the non-zero basis functions, equation 4.24 is rewritten as

\[
C(t) = \sum_{i=a-p}^{a} P_i^h B_{i,p}(t)
\]  \(\text{(6.27)}\)
for the valid knot span \([t_a, t_{a+1}]\). Or for a surface

\[
S(u, v) = \sum_{i=a-p}^{a} \sum_{j=b-p}^{b} P_{i,j}^h B_{i,p}(u) B_{j,q}(v)
\]  

(6.28)

given the valid knot spans \([u_a, u_{a+1}]\) and \([v_b, v_{b+1}]\).

This scheme consequently requires that the valid span index \(a\) is found, which can easily be done given a parameter value \(t\) by iterating through the knot vector until arriving at the index \(a\) for which \(t_a \leq t < t_{a+1}\). If no such value exists, then \(t\) lies outside of the defined parameter interval.

### 6.6.1 Cox De-Boor Items

Considering the local support property of NURBS surfaces and given that the knot vectors, degrees and number of control points for the surface are known in advance, some parts of the Cox De-Boor recursive formula can be computed in advance and stored in a separate variable to later be used for the surface evaluation. These precomputed items are called Cox De-Boor items or CDBItems for short. The method originates with Abert, Geimer and Mulle [30]

The first step for precomputing the CDBItems is to rewrite the Cox De-Boor formula into a form that allows for precomputation of invariant terms. The formula is rewritten as

\[
B_{i,p}(t) = c_1(t - t_i)B_{i,p-1}(t) + c_2(t_{i+p+1} - t)B_{i+1,p-1}(t)
\]  

(6.29)

Where

\[
c_1 = \frac{1}{t_{i+p} - t_i}
\]  

(6.30)

\[
c_2 = -\frac{1}{t_{i+p-1} - t_{i+1}}
\]  

(6.31)

The formula can be further substituted to give

\[
B_{i,p}(t) = (c_2 t + c_4)B_{i+1,p-1}(t) + (c_1 t + c_3)B_{i,p-1}(t)
\]  

(6.32)

Where \(c_3 = -t_i c_1\) and \(c_4 = -c_2 t_{i+p+1}\)

These four values \(c_i, i \in [1,4]\) are the CDBItems computed during a preprocessing step. The values are precomputed for each of the basis-functions \(B_{i,p}(t)\) in both \(u\) and \(v\) direction except for \(B_{i,0}(t)\) which is known to be equal to one for both \(u\) and \(v\) considering the local support property of the surface.
At runtime a basis-function \( B_{i,p}(t) \) can be calculated as long as the parameter \( t \) as well as the two lower degree basis functions \( B_{i+1,p-1}(t) \) and \( B_{i,p-1}(t) \) are known using the CDBItems.

### 6.6.2 Basis-Function Cache

There is a further optimization presented by Abert et.al. in [30] that can be made to the surface evaluation of NURBS surfaces. As mentioned, each basis-function \( B_{i,p}(t) \) has a set of lower degree basis functions on which it depends. Looking at the dependencies for the basis-functions in the Cox De-Boor formula, it becomes clear that many basis-functions can end up sharing the same dependencies. A naïve approach would calculate these dependencies each time a basis-function is calculated, however a faster alternative is to pre-calculate these dependencies and store them in a cache that can be accessed by the surface evaluator.

Since the basis-functions depend on the parameter values \( u, v \), they cannot be computed during the preprocessing step, rather a cache is computed in both directions for every new \( u, v \) value before performing the surface evaluation.

Since every basis-function depends on two lower non-zero basis functions, the lower order basis-functions are calculated first starting from the 0th degree basis function which is known to be equal to one. As an example, given a cubic NURBS surface with 16 control points and the knot vectors \([0 0 0 0 1 1 1 1]\) in both directions, the basis-functions in a single direction that need to be evaluated are \( B_{0,3}(t), B_{1,3}(t), B_{2,3}(t) \) and \( B_{3,3}(t) \). The non-zero dependencies of these basis functions can be represented with the following triangular scheme.

\[
\begin{array}{cccc}
B_{0,3}(t) & B_{1,3}(t) & B_{2,3}(t) & B_{3,3}(t) \\
B_{1,2}(t) & B_{2,2}(t) & B_{3,2}(t) & \\
B_{2,1}(t) & B_{3,1}(t) & & \\
B_{3,0}(t) & & & \\
\end{array}
\]

Since \( t \) is within the valid parameter interval, \( B_{3,0}(t) \) is known to be equal to one. The rest of the basis-functions are calculated using equation 6.32 with the CDBItems for each basis function in the order

\[
\begin{array}{cccc}
9 & 8 & 7 & 6 \\
5 & 4 & 3 & \\
2 & 1 & & \\
0 & & & \\
\end{array}
\]

The lowest degree basis function, which is known to be equal to one is calculated first. Second, the two higher degree basis functions that depend on the lowest basis function are calculated from right to left, then the three higher degree basis functions are calculated bases on the two lower ones, and so on, until the \( p + 1 \) basis functions needed for the surface evaluation are calculated. The same method works for the derivative calculation as well.
6.7 NURBS Subdivision

The Oslo algorithm outlined in section 4.5.5 can be used to recursively subdivide a NURBS curve or surface as per a method described by Peterson in [42] originally used for tessellating NURBS.

Splitting a NURBS curve using the Oslo algorithm is done by adding new knots of the same value in the knot vector, where the number of new knots inserted increases the multiplicity of that value to the degree of the curve plus one. Then, using the Oslo algorithm (equation 4.38), a new set of control points describing the exact same curve can be determined.

As an example, taking the knot vector \([0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1]\) which describes a cubic curve with four control points, knots are inserted in the middle with multiplicity four which gives the new knot vector \([0 \ 0 \ 0 \ 0.5 \ 0.5 \ 0.5 \ 1 \ 1 \ 1 \ 1]\). Since the number of elements in a knot vector is equal to \(p + n + 1\) for a curve or surface with \(n\) control points and degree \(p\), this new knot vector describes a curve with eight control points where the first four control points have local support within the interval \(t \in [0, 0.5]\) and the last four within \(t \in [0.5, 1]\). The surface can then easily be split by creating two new curves, where the first curve is described by the first four control points with the knot vector \([0 \ 0 \ 0 \ 0.5 \ 0.5 \ 0.5 \ 0.5]\) and the second curve is described by the last four control points with the knot vector \([0.5 \ 0.5 \ 0.5 \ 1 \ 1 \ 1 \ 1]\). Note that for the new set of eight control points created by the initial knot insertion in this particular case, the two middle control points overlap.

6.7.1 NURBS Flatness Test

The process of determining whether or not a NURBS surface is straight in Peterson’s subdivision method [42] is done by first determining if each of the curves in both \(u\) and \(v\) direction that compose the surface are straight and also making sure that the surface is not twisted by checking if the corners of the surface are coplanar.

To test the straightness of a NURBS curve a line is created between the first control point and another control point that is significantly far from the first.

\[
\|P_i - P_0\| > \varepsilon \quad (6.33)
\]

Once such a line is found it is stored in a separate vector \(T = P_i - P_0\). Then the curves are iterated over again where the flatness of each successive line is determined in relation to \(T\) by

\[
\| (P_i - P_0) \times T \| \quad (6.34)
\]

As long as the value is smaller than some user defined threshold for each successive pair of control points the curve is considered to be flat enough.

Once the flatness of each curve in both directions of the surface has been determined a final check is made to make sure that the control points are coplanar in order to assure that the surface is not twisted.
A plane is created using the cross product of two adjacent edges of the surface

\[ P = (C_{0,1} - C_{0,0}) \times (C_{1,0} - C_{0,0}) \]  

(6.35)

where \( C_{i,j} \) is a control point that is one of the corners on the surface.

The distance from the plane \( P \) to the fourth corner \( C_{1,1} \) is then given by

\[ \|C_{1,1} - P\| \]  

(6.36)

If the distance is smaller than a user defined value \( \varepsilon \) then the surface is not significantly twisted and subdivision can be halted if the curves are also reasonably flat.

7. Implementation

7.1 Implicit Surfaces
An implicit sphere and implicit cylinder are implemented according to the methods described in sections 5.1 and 5.2. Both are created with the SIMD library in order to increase performance.

7.2 Parametric Surfaces

7.2.1 Bezier Surface Preprocessing
The preprocessing step for the Bézier surface consists of loading the control points or the model, determining the bounds of the model for Embree’s bounding volume hierarchy, subdividing each of the surfaces loaded, creating a bounding volume hierarchy out of the subdivided patch and calculating the inner product \([N][CP][N]\).

The control points for the surface can either be defined manually by an array of sixteen three-dimensional vectors or by loading a model containing the control points. Two types of files are supported for loading the Bézier models. One is a custom file type consisting of a list of vertices corresponding to each of the control points along with a list of indices, with sixteen indices per line, where each line represents a patch in the model and each index in the line corresponds to one of the vertices. The model is loaded by first loading all of the vertices and then for each of the patches a new Bézier patch model is instantiated along with a call for the surface to be preprocessed. Another type of file is also supported, which is the Wavefront OBJ file type. The OBJ file type supports the representation of rational and non-rational b-spline surfaces. The structure of a b-spline OBJ file is similar to that of the custom file in that there is a list of vertices corresponding to the control points along with lines of sixteen indices per line corresponding to the patches. The file also contains the knot vectors and degrees of each of the surfaces. These are ignored in the case of loading Bézier patches since all of the knot vectors are pre-set to eight length vectors with a multiplicity of four for the first and last knot vector and all of the degrees are set to three, which as was mentioned in section 4.5.3, describes a cubic Bézier curve.

The bounds can easily be determined for each patch by iterating through all of the control points, finding the minimum and maximum x, y and z values and storing them in
Embree’s `bounds_o` type, which is used by Embree to determine the bound of the bounding box.

Subdivision is done using De Casteljau’s algorithm as described in section 4.4.3. The subdivision depth along with the flatness tolerance are defined when instantiating the Bézier surface object. The subdivision continues recursively until either the maximum depth is reached or the surface is reasonably flat after which a leaf node for the bounding volume hierarchy is created from the subdivided surface. Code example 1 gives the code for determining the flatness of a single Bézier curve. Along with determining new control points for the leaf node, the initial guesses that are to be stored in each leaf node are also determined. Given an initial value of \( u, v = 0.5 \) along with the parametric width and height which are each set to \( w, h = 1 \) the new initial guesses are determined by the recursive formula

\[
\text{split}(u_m, v_m, w_m, h_m) = \left((u_m - \frac{w_m}{4}, v_m, \frac{w_m}{2}, h_m), (u_m + \frac{w_m}{4}, v_m, \frac{w_m}{2}, h_m)\right)
\]

(7.1)

for the vertical split in \( u \)'s direction and

\[
\text{split}(u_m, v_m, w_m, h_m) = \left((u_m, v_m - \frac{h_m}{4}, w_m, \frac{h_m}{2}), (u_m, v_m + \frac{h_m}{4}, w_m, \frac{h_m}{2})\right)
\]

(7.2)

for a horizontal split in \( v \)'s direction.

```cpp
float Bezier::flatness (vec3 P0, vec3 P1, vec3 P2, vec3 P3) {
    float ux = 3.0f * P1.x - 2.0f * P0.x - P3.x; ux *= ux;
    float uy = 3.0f * P1.y - 2.0f * P0.y - P3.y; uy *= uy;
    float uz = 3.0f * P1.z - 2.0f * P0.z - P3.z; uz *= uz;

    float vx = 3.0f * P2.x - P0.x - 2.0f * P3.x; vx *= vx;
    float vy = 3.0f * P2.y - P0.y - 2.0f * P3.y; vy *= vy;
    float vz = 3.0f * P2.z - P0.z - 2.0f * P3.z; vz *= vz;

    ux = max(ux, vx); uy = max(uy, vy); uz = max(uz, vz);

    return ux + uy + uz;
}
```

Code example 1 Calculating the flatness of a Bézier curve given four control points.

### 7.2.2 NURBS Surface Preprocessing

Preprocessing the NURBS surface requires the following five steps
• Load the control points, knot vectors and degrees either from user defined values or from files containing several surfaces.
• Instantiate one surface cache structure per thread used by the system for both $u$ and $v$ direction.
• Calculate the Cox De-Boor items based on the knot vector.
• Subdivide the surface until the maximum subdivision depth is reached or the new surfaces are flat enough based on a user defined tolerance.
• Generate the bounding volume hierarchy from the subdivided surface.

The control points, knot vector and degrees can be manually chosen and placed in the NURBS object structure for any degree and any number of control points in any direction. It is also possible to load the Bézier patches from the custom file type described in section 7.2.1 by setting the degrees to three and the knot vectors to $[0 \ 0 \ 0 \ 1 \ 1 \ 1]$. Alternatively, NURBS surfaces can be read from OBJ files.

Since the size of the elements of the surface cache for both the surface and the derivatives is equal to the degree of the surface plus one they need to be dynamically instantiated on the heap depending on the degree of the surface. Allocating an object on the heap for each surface evaluation slows down performance so the surface caches are allocated during the preprocessing step and stored in the NURBS object since the degree is known beforehand. However, since the system uses multiple threads to perform the intersection a surface cache needs to be created for each thread so that there are no access conflicts and so as to avoid using critical sections which can also slow down performance. Each thread has an identity number which is used as the index for retrieving and storing values in the corresponding surface cache.

After allocating the surface caches, the Cox De-Boor items or CDBItems are determined from the knot vector using the algorithm in code example 2.

```cpp
CDBitemsU = new float**[degreeU];
for (int d = 1; d <= degreeU; d++) {
    CDBitemsU[d - 1] = new float*[cpNumU];
    for (int i = cpNumU - 1; i >= degreeU - d; i--) {
        CDBitemsU[d - 1][cpNumU - i - 1] = new float[4];
        CDBitemsU[d - 1][cpNumU - i - 1][0] = knotU[i + d] - knotU[i] == 0 ? 0 : 1 / (knotU[i + d] - knotU[i]);
        CDBitemsU[d - 1][cpNumU - i - 1][1] = knotU[i + d + 1] - knotU[i + 1];
        CDBitemsU[d - 1][cpNumU - i - 1][2] = (-CDBitemsU[d - 1][cpNumU - i - 1][0] * knotU[i]);
        CDBitemsU[d - 1][cpNumU - i - 1][3] = (-CDBitemsU[d - 1][cpNumU - i - 1][1] * knotU[i + d + 1]);
    }
}
```

**Code example 2 Calculating the CDBItems in u’s direction**

The subdivision scheme used for NURBS surfaces is based on a method used for tessellating NURBS surfaces found in [42] which can also be used for general subdivision. Basically the method uses the Oslo Algorithm (outlined in section 4.5.5) in order to split the surface along either direction. The algorithm starts by inserting multiple knots at a parameter value in the middle of the knot vector so that it has multiplicity $k = degree + 1$. The new control points that described the exact same
surface are then determined using the Oslo algorithm. Once the new control points are determined, the surface is simply split by creating two new sets of control points ending and beginning at the control point determined by the knot span where the multiplicity was set. The flatness is then determined by the method described in section 6.7.1.

The bounding volume is generated in the exact same way as for the Bézier surface. The only difference is the generation of the initial guesses for the leaf nodes. The initial guesses for the NURBS leaf nodes is done by dividing the sum of the first and last elements of the valid knot span by two.

\[ u = \frac{u_p + u_n}{2} \]  

(7.3)

where \( p \) is the degree and \( n \) is the number of control points in \( u \)'s direction. The value of \( v \) remains the same as the divided surface when dividing in \( u \)'s direction and vice versa.

### 7.2.3 Bounding volume hierarchy

Since Embree automatically creates a bounding volume hierarchy for a given scene when calling the `RTCcommit` function this is used for the creation of the bounding volume hierarchy of each of the patches as well. A user defined geometry is created out of the control points of the leaf nodes determined by the subdivision process where the bounds of the user defined geometry are determined in the same fashion as they were determined for the patch. These user-defined geometries are stored in an `RTCScene` type one by one until the subdivision has ended. Once all of the leaf nodes have been stored in the scene a call to commit the scene is performed after which Embree automatically creates a bounding volume hierarchy out of the bounds defined for each of the leaf nodes. The leaf nodes also contain the initial guesses for the Newton iteration.

Since the computational expense of the parametric surfaces is high relative to traversing the bounding volumes, a more novel traversal scheme is used. Instead of testing for intersection for each leaf node that the ray intersects, all of the leaf nodes that the ray intersects are stored in a cache. The cache is then sorted based on the distance from the leaf node to the ray’s origin and the intersection tests are conducted starting with the leaf node closest to the ray’s origin. If no intersection is found, or not all of the rays in the packet have found an intersection, the intersection tests are continued down the leaf node cache until either an intersection is found or there are no more leaf nodes.

![Figure 6 A Bézier surface subdivided to three different levels using Embree's bounding volume generation. The boxes are colored after the initial guesses contained in the leaf nodes, with red being in v's direction and green in u's direction.](image)

### 7.2.4 Intersection Testing

The Newtonian method described in section 6.3 is used for determining the intersection of a ray and a Bézier or NURBS surface. The ray is as mentioned first passed to the
bounding volume hierarchy in order to generate the initial guesses needed for the method to converge fast and reliably. The intersection testing uses the SIMD library from Embree's source code. This increases the performance of the intersection testing since eight rays can be tested simultaneously. However the entire ray packet needs to be taken through all of the iterations until the testing is done for each of the rays in the packet.

7.2.5 Bézier Surface Evaluation
The Bézier surface evaluator is implemented using the matrix representation of Bézier surfaces (section 4.4.6) and calculated according to the method described in section 6.4.

7.2.6 NURBS Surface Evaluation
The evaluation of the NURBS surface uses the optimized method described in section 6.6 to avoid calculating basis functions that are known to not contribute within the evaluated knot span. The basis function cache (described in section 6.6.2) is also calculated for each new parameter value with the precomputed CDBItems.

For NURBS surfaces the parameter values are checked to see if they are within the valid knot span and passed to the surface evaluator if they are. For packets of rays the initial guesses in a single packet may lie in different knot spans, in those cases it is not possible to perform the surface evaluation using SIMD. The values are checked to see if the values are all within the same knot span. If they are then the packet is flagged as coherent and is passed to the SIMD surface evaluator, if not then the packet is split and each element in the packet is calculated one by one.

8. Results

8.1 Test System
All performance tests were performed with an Intel Core i7-5960X with 8 physical and 16 logical cores. For the performance evaluation, the system was compiled with Visual Studio. For memory consumption a build using Xcode was used.

8.2 Test Setup
In order to properly test the system, a benchmarking tool was developed. When the tool is enabled, several aspects of the system are measured and stored. When a specified amount of frames have been computed a call to the benchmarking tool causes it to calculate averages from the accumulated data. Table 4 gives some of the values that are measured by the benchmarking tool and presented in this section.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg fps</td>
<td>The average framerate over the specified amount of frames</td>
</tr>
<tr>
<td>Avg ms</td>
<td>The average time in milliseconds to render a single frame</td>
</tr>
<tr>
<td>Screen fill</td>
<td>The number of rays with a positive intersection test calculated as a percentage of the amount of primary rays.</td>
</tr>
</tbody>
</table>
Besides the values in table 4, the framerate is measured for each second that passes and stored in the benchmarking tool.

All of the performance tests are measured at a resolution of 1920x1080 where the camera revolves around the models for 60 seconds in order to gather accurate average values and present the performance difference when the screen fill is larger or smaller.

Memory consumption is determined by measuring the amount of persistent bytes that are allocated after a call to the model’s loading function using Xcode’s allocation analyzer tool.

8.3 Polygonal Meshes

8.3.1 Image Quality

Generally, the image quality of polygonal meshes depends on the amount of faces present in the model. Figures 7 and 8 show tessellated versions of the models used in this thesis, where the models have been tessellated from their original NURBS representation using various levels of tessellation tolerance, resulting in a various amount of triangles.

Figure 7 The Teapot model tessellated with different levels of tolerance. From left to right the triangle count is 9,793, 34,514, 299,035 and 2,839,344 triangles.

Figure 8 The Killaroo model tessellated with different tolerances from the original NURBS representation. From left to right the triangle count is 91,902, 205,131, 529,127 and 4,611,833 triangles.

For the sphere and cylinder, the meshes are instead subdivided from an original mesh representation and are shown in figures 9 and 10.

Figure 9 A mesh representation of the sphere. The model is subdivided from an originally coarse representation in order to give models with different amounts of triangles. From left to right the triangle count is 512, 1,984, 7,936 and 31,744 triangles.
8.3.3 Memory Consumption

The memory consumed by polygonal meshes scales based on the amount of triangles in the mesh. Table 5 shows the amount of megabytes (MB) consumed by the four polygonal meshes evaluated at various degrees of tessellation. Any memory allocated that was later freed is ignored, the only memory that is measured is the amount of persistent bytes allocated by the mesh loading method.

Table 5 Memory consumption for meshes with a varying amount of triangles. The memory consumed by polygonal models depends on the amount of triangles that constitute the model.

<table>
<thead>
<tr>
<th></th>
<th>Sphere</th>
<th>Cylinder</th>
<th>Teapot</th>
<th>Killaroo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>MB</td>
<td>Triangles</td>
<td>MB</td>
<td>Triangles</td>
</tr>
<tr>
<td>512</td>
<td>0.088</td>
<td>124</td>
<td>0.014</td>
<td>9,793</td>
</tr>
<tr>
<td>1,984</td>
<td>0.346</td>
<td>744</td>
<td>0.088</td>
<td>34,514</td>
</tr>
<tr>
<td>7,936</td>
<td>1.35</td>
<td>2,976</td>
<td>0.346</td>
<td>299,035</td>
</tr>
<tr>
<td>31744</td>
<td>5.38</td>
<td>11,904</td>
<td>1.35</td>
<td>2,839,344</td>
</tr>
</tbody>
</table>

8.4 Implicit Surfaces

8.4.1 Image Quality

Figures 11 and 12 show the two implicit surfaces implemented in this thesis. Ray-tracing algebraic implicit surfaces is as can be seen generally very robust with little to no artefacts. There are also no variables to consider that improve or diminish the quality of the objects. Quality is also maintained regardless of how close the camera is to the surface.
Figure 11 The implicit version of the sphere model. For implicit surfaces there are no variables that alter the quality of the models, rather the model remains smooth and of good quality regardless of any variables.

Figure 12 The implicit version of the cylinder model. As with the sphere there are no variables that need to be tuned in order to increase the quality.

8.4.2 Performance
Considering the symmetrical shape of the implicit surfaces evaluated, instead of comparing single implicit surfaces with polygonal counterparts the models are arranged in a 40 x 40 grid with a total of 1600 objects in each grid that the camera revolves around while performance measurements are collected. Figures 13, 14 and 15 give the results for spheres, cylinders and a mixture of both respectively.

Figure 13 A performance comparison of implicit spheres arranged in a 40 x 40 grid and polygonal meshes of the same size also placed in a 40 x 40 grid. The polygonal meshes each consist of 512 triangles. The camera is spun around the grid and the frames per-second are measured at set intervals.
Table 6 Additional information for the sphere grid test. The data presented in this table shows average values accumulated throughout the duration of the performance test.

<table>
<thead>
<tr>
<th>Sphere grid test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Implicit sphere</td>
<td>Sphere mesh</td>
</tr>
<tr>
<td>Triangles</td>
<td>n/a</td>
<td>512</td>
</tr>
<tr>
<td>Avg fps</td>
<td>29.7131</td>
<td>22.2721</td>
</tr>
<tr>
<td>Avg ms</td>
<td>31.7999</td>
<td>41.4735</td>
</tr>
<tr>
<td>Screen fill</td>
<td>29.0095%</td>
<td>28.7672%</td>
</tr>
</tbody>
</table>

Figure 14 The grid test performed on the implicit cylinders as well as polygonal approximations consisting of 744 triangles. The camera revolves around the grid while performance measurements are stored.

Table 7 Additional details of the grid test for cylinder models. The values are averages of the values computed throughout the tests.

<table>
<thead>
<tr>
<th>Cylinder grid test</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Implicit cylinder</td>
<td>Cylinder mesh</td>
</tr>
<tr>
<td>Triangles</td>
<td>n/a</td>
<td>744</td>
</tr>
<tr>
<td>Avg fps</td>
<td>29.9921</td>
<td>26.8887</td>
</tr>
<tr>
<td>Avg ms</td>
<td>31.5488</td>
<td>34.856</td>
</tr>
<tr>
<td>Screen fill</td>
<td>22.4149%</td>
<td>23.3229%</td>
</tr>
</tbody>
</table>
The grid test performed with a mix of implicit cylinders and spheres compared with a mix of polygonal meshes of spheres and cylinders. The objects are placed one after another in a 40 x 40 grid. The polygonal sphere consists of 512 triangles while the polygonal cylinder consists of 744 triangles.

Table 8 Additional details for the grid test conducted on a mix of the objects. The values are averages determined from the values calculated throughout the test in Figure 15.

<table>
<thead>
<tr>
<th>Types</th>
<th>Mixed implicits</th>
<th>Mixed meshes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>n/a</td>
<td>744 and 512</td>
</tr>
<tr>
<td>Avg fps</td>
<td>25.9233</td>
<td>24.7169</td>
</tr>
<tr>
<td>Avg ms</td>
<td>36.0923</td>
<td>37.6894</td>
</tr>
<tr>
<td>Screen fill</td>
<td>25.5392%</td>
<td>25.4614%</td>
</tr>
</tbody>
</table>

As can be seen, implicit surfaces generally outperform even the simplest polygonal approximations of the same objects. The performance becomes relatively close when the types are mixed as can be seen in table 8, however the meshes involved in each test aren’t close to the same quality of the implicit surfaces.

8.4.3 Memory Consumption
The memory consumption of the implicit surfaces implemented remains the same regardless of any variables. The sphere object is 32 bytes total and the cylinder is 48 bytes. However, for both objects a baseline of 2.52 kilobytes is allocated by the library when adding a new object. Compared with the memory results for the meshes in table 5, implicit representations clearly have an advantage when it comes to memory consumption.

8.5 Bézier Surfaces

8.5.1 Image Quality
Figures 16 and 17 show the image quality at the subdivision level and tolerance that has been found to produce acceptable image quality while minimizing the amount of bounding boxes created.
The image quality and reduction of most artefacts heavily depends on the subdivision depth and tolerance. Figure 18 shows the increase in image quality when increasing the subdivision depth for the teapot.

8.5.2 Performance
Figures 19 and 20 show the Bézier versions of the Teapot and Killaroo models compared with tessellated versions of the same models. As can be seen the Bézier versions generally perform slower than the polygonal meshes, even when a very large amount of triangles are involved.
Figure 19 Performance evaluation of the Bézier Teapot compared with polygonal meshes of the same model tessellated with different tolerances in order to produce models with different amount of triangles.

Table 9 Additional information for the test represented in Figure 19. The values are averages calculated from the values in Figure 19.

<table>
<thead>
<tr>
<th></th>
<th>Triangle Mesh</th>
<th>Bézier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles / Leaf Nodes</td>
<td>34 514</td>
<td>2 839 344</td>
</tr>
<tr>
<td>Avg FPS</td>
<td>27.0482</td>
<td>19.9227</td>
</tr>
<tr>
<td>Avg ms</td>
<td>34.6107</td>
<td>46.1823</td>
</tr>
<tr>
<td>Screen fill</td>
<td>23.907%</td>
<td>24.7349%</td>
</tr>
</tbody>
</table>

Figure 20 Performance comparison between the Bézier version of the Killaroo model with polygonal meshes of the same model tessellated with different tolerances. The test is performed by revolving the camera around the models over 60 seconds and measuring the framerate at set intervals.
Table 10 Additional information for the test presented in Figure 20. The values are averages calculated from the values collected during the test.

<table>
<thead>
<tr>
<th></th>
<th>Triangle Mesh</th>
<th>Bézier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles / Leaf nodes</td>
<td>205 131</td>
<td>4 611 833</td>
</tr>
<tr>
<td>Avg fps</td>
<td>32.2853</td>
<td>29.0094</td>
</tr>
<tr>
<td>Avg ms</td>
<td>29.2065</td>
<td>32.4768</td>
</tr>
<tr>
<td>Screen fill</td>
<td>12.5431%</td>
<td>12.5194%</td>
</tr>
</tbody>
</table>

### 8.5.3 Memory Consumption

The memory consumed by Bézier surfaces depends on both the subdivision depth and the amount of patches in the model. For the teapot model evaluated in section 8.5.2 the memory consumed is 2.98 megabytes with 14 376 leaf nodes. For the Killaroo model with 115 782 leaf nodes the memory consumed is 98.99 megabytes. Table 11 gives the memory consumed for both the teapot and the Killaroo model at various levels of subdivision. As can be seen with the Killaroo model the amount of bounding boxes increases drastically relative to the increase in the level of subdivision.

Table 11: The memory consumption for the Bézier models with different levels of subdivision. For the teapot the memory consumption is very low and does not scale greatly with an increase in subdivision depth. On the other hand the more organic looking and curved Killaroo model produces many more leaf nodes given the large number of patches in the model

<table>
<thead>
<tr>
<th>Subdivision level</th>
<th>Leaf nodes</th>
<th>MB</th>
<th>Subdivision level</th>
<th>Leaf nodes</th>
<th>MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>64</td>
<td>1.23</td>
<td>0</td>
<td>22,928</td>
<td>81.00</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>1.38</td>
<td>4</td>
<td>366,848</td>
<td>133.48</td>
</tr>
<tr>
<td>8</td>
<td>16384</td>
<td>3.72</td>
<td>8</td>
<td>5,869,568</td>
<td>973.12</td>
</tr>
</tbody>
</table>

### 8.6 NURBS

#### 8.6.1 Image Quality

As with the Bézier surface versions, the quality of the NURBS versions of the models generally depends on the subdivision depth and tolerance. Figures 21 and 22 show the quality of the models at the subdivision and tolerance levels chosen for the performance tests.
Figure 21 The NURBS version of the Teapot model with 16,384 bounding volumes. The subdivision depth and tolerance is set to a level that produces acceptable image quality.

Figure 22 The NURBS version of the Killaroo model. The subdivision depth and tolerance is set to a level that produces acceptable image quality while maintaining a low amount of bounding volumes. The number of leaf nodes is 180,980.

Figure 23 shows the increase in image quality for an increase in subdivision depth.

Figure 23 The image quality of the NURBS teapot relative to the amount of bounding boxes in the model. From left to right the amount of leaf nodes is 64, 256 and 4096

8.6.2 Performance
Figures 24 and 25 show comparisons between NURBS models of the Teapot and Killaroo along with tessellated versions of the same models. As can be seen there's a larger performance disparity between the NURBS and tessellated Killaroo models than for the teapot model. The Killaroo model is very detailed with lots of curves and therefore requires a much larger amount of bounding volumes. Each of the patches that make up the model are also very large and complex, further decreasing the performance.
A performance comparison of the NURBS Teapot compared with tessellated versions of the same model. The camera rotates around each of the models during 60 seconds while the framerate is recorded during set intervals.

Table 12 Additional information for the NURBS Teapot performance test in Figure 24. The values are averages of the data calculated throughout the test.

<table>
<thead>
<tr>
<th>NURBS Teapot compared with triangle meshes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Triangles / Leaf nodes</td>
</tr>
<tr>
<td>Avg fps</td>
</tr>
<tr>
<td>Avg ms</td>
</tr>
<tr>
<td>Screen fill</td>
</tr>
</tbody>
</table>

The NURBS version of the Killaroo model compared with tessellated counterparts. The camera here is also revolved around the models while framerate is computed at set intervals.

Figure 25 The NURBS version of the Killaroo model compared with tessellated counterparts. The camera here is also revolved around the models while framerate is computed at set intervals.
Table 13 Complementary information for the NURBS Killaroo performance test in Figure 25. The values are averages computed from the values in Figure 13 and others.

<table>
<thead>
<tr>
<th>NURBS Killaroo compared with triangle meshes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>Triangles / Leaf nodes</td>
</tr>
<tr>
<td>Avg fps</td>
</tr>
<tr>
<td>Avg ms</td>
</tr>
<tr>
<td>Screen fill</td>
</tr>
</tbody>
</table>

8.6.3 Memory Consumption
The memory consumed by the teapot model with 16 384 leaf nodes evaluated in section 8.6.2 is 3.75 megabytes and for the Killaroo model with 180 980 leaf nodes is 31.69 megabytes. Table 14 gives the memory consumption relative to the amount of leaf nodes in the model. Comparing the results with those for the Bézier surfaces it can be seen that while memory consumption is greater for the simpler Teapot model, the memory consumed for the Killaroo model is much less. This is due to the fact that while a single NURBS patch consumes more memory than a single Bézier patch, the Bézier Killaroo model requires a very large amount of patches in comparison with the 89 required for the NURBS representation.

Table 14 Memory consumption of the NURBS models at different levels of subdivision. As can be seen the memory consumption for the NURBS Killaroo model remains fairly small even with large levels of subdivision since only 89 patches are needed to represent the model

<table>
<thead>
<tr>
<th></th>
<th>Teapot (32 patches)</th>
<th>Killaroo (89 patches)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Subdivision level</td>
<td>Leaf nodes</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>1.26</td>
</tr>
<tr>
<td>4</td>
<td>1024</td>
<td>1.40</td>
</tr>
<tr>
<td>8</td>
<td>16384</td>
<td>3.75</td>
</tr>
</tbody>
</table>

9. Discussion
For implicit surfaces, frame-rates are generally higher, even when compared with fairly crude polygonal approximations. Implicit surfaces also maintain the same image quality regardless of how close the camera is to the surface, while polygonal meshes do not. Memory usage for implicit surfaces is also lower even when compared to fairly low polygonal approximations. The image quality of implicit surfaces is generally very robust and there are no variables that need to be fine-tuned for the image quality to be good, at least for the types evaluated in this thesis. Clearly the frame-rates for the implicit surfaces evaluated are within interactive frame-rates.

Although frame-rates are generally lower for parametric surfaces compared to polygonal meshes, even when a very large amount of triangles are used, the values are generally within interactive frame-rates, even at a resolution of 1920 x 1080. Bézier surfaces can come fairly close to the performance of their polygonal counterparts.
(figures 19 and 20) for both of the models evaluated, while NURBS are both much slower and have a larger performance disparity for an increase in the complexity of the model as can be seen in figures 24 and 25. The reason for the increased performance disparity for NURBS surfaces is likely due to the fact that ray packets need to be split and evaluated one by one when the rays are in different knot spans which is a likely occurrence for large complex patches.

A major advantage that parametric surfaces have over their polygonal counterparts is lower memory usage. The memory usage is not only generally lower for a number of leaf nodes that produce acceptable quality, the memory usage also does not need to increase for better image quality when the camera is close to the surface. Bézier and NURBS surfaces maintain the same level of smoothness regardless of how close the camera is to the surface, while polygonal meshes would need to increase the number of triangles relative to how close the camera is to the model in order to have the same property. Increasing the number of triangles would however also increase the amount of information in the model which would increase memory usage.

Good image quality is achievable for Bézier surfaces even with a relatively low amount of leaf nodes. Generally the teapot model can be represented without noticeable artifacts for both NURBS and Bézier surfaces even with a fairly low amount of bounding volumes. For the Killaroo model, good image quality without many artifacts can be achieved for Bézier surfaces while the NURBS version has some artifacts that are not necessarily removed when increasing the amount of leaf nodes. This is also likely due to the large complex patches that constitute the model.

Of course one of the major advantages of ray tracing is the ability to simulate light and add additional effects such as shadows, reflection and refraction. For NURBS surfaces it is doubtful that interactive frame-rates can be maintained for complex objects with additional effects, at least at the resolution evaluated.

9.1 Related Research
There have not been many recent attempts at ray tracing parametric surfaces. In [30] the same NURBS ray tracing method was used and tested on a dual-code PowerMac G5 with some of the same models tested here achieving a maximum of 7 frames per second for the teapot. In [32] a GPU implementation was tested on a GeForce GTX 295 which achieved up to 30 frames-per second for the teapot model. In a 2008 paper [43] up to 40 frames-per second for the teapot model and 30 frames-per second for the Killaroo model was achieved on an 8-core CPU.

As for Bézier surfaces, in a Master's thesis from 2006 [44], a GPU implementation on a GeForce 6800 achieved a maximum of two frames per second. In [31] a CPU implementation was able to achieve around 6 frames per second for the teapot model.

As for other direct comparisons between parametric surfaces and polygonal meshes. In a 2004 study [45] a comparative study between Bézier surface, Loop subdivision surfaces and triangle meshes was performed with the Bézier surface having about 50% of the speed of triangle meshes. No NURBS were tested in the study unfortunately, and other studies directly comparing parametric surfaces with triangle meshes have not been found.
10. Conclusions

A ray tracing system able to handle Bézier and NURBS surfaces as well as implicit surfaces was created using Embree in order to determine the performance of each in comparison with polygonal models representing the same objects. Performance close to polygonal counterparts for Bézier surfaces was achieved, with NURBS being slower than Bézier surfaces but still within interactive frame-rates.

The image quality for parametric surfaces can vary greatly depending on the subdivision implementation. The parameters that determine the level of subdivision are dependent on the model and need to be fine-tuned in order to optimize results. Having a level of subdivision that is very high results in high image quality but lowers performance. It is therefore crucial to have a good adaptive subdivision scheme for both surface types.

Memory consumption was shown to be lower for all of the non-polygonal representations implemented and evaluated giving them a clear advantage over polygonal meshes. The memory consumption is not only lower but also does not need to be increased when the camera moves closer to the surfaces since the same level of smoothness is maintained for the evaluated geometries.

The ray tracing system was written with a SIMD library in order to allow for computation of ray packets with eight rays per packet. The SIMD aspect of the implementation gave a large increase in performance and allowed for the representations implemented to reach interactive frame-rates, even at a high resolution making them suitable for interactive ray tracing, at least without many additional lighting effects.

10.1 Future work

Although it is possible to achieve interactive frame-rates that come close to the performance of triangle meshes with both Bézier surfaces and NURBS surfaces there are still problems relating to non-convergence with Newton’s method. A study implementing the second order derivative method [33] for parametric surfaces would be of interest considering that it is more reliable than Newton’s method according to the authors.

The use of ray streams, where the size of the ray packet is decreased as rays in the packet are determined to not intersect the surface would be of interest since ray packets need to be taken through all of the newton iterations for parametric surfaces if even one is slow to converge. Ray streams could also be used to fix the problem with incoherent ray packets needing to be split for NURBS surface evaluation.

A study that implements additional lighting and other visual effects would also be of great interest.

As for implicit surfaces, a more general algorithm such as the Sphere-tracing method [18] or the method based on interval arithmetic by Knoll et.al. [46] would be of great interest. Although the algebraic surfaces implemented in this paper were considerably faster than identical triangle meshes, a more general algorithm would likely be slower whilst also being more useful for modeling general implicit surfaces.
Bibliography

[38] O. P. Abert, "Interactive ray tracing of NURBS surfaces by using SIMD instructions and the GPU in parallel.," Diss. Nanyang Technological University, 2005.