Subdivision Surfaces for use in Cut Finite Element Methods

Kjetil Justnes

Thesis submitted for the degree of
Master in Computer Science: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018
Subdivision Surfaces for use in Cut Finite Element Methods

Kjetil Justnes
Abstract

The finite element method has for some time been a popular choice for approximating PDEs over a geometry domain. We usually use a FEM mesh that matches the geometry description. By FEM mesh we mean the mesh where we find the solution to the PDE. In order to use finite element method where the geometry description does not match the FEM mesh, we need a technique to modify the FEM around the geometry description. An essential part of such techniques is to find the intersection points between the geometry description and the FEM mesh. We say that FEM mesh is cut when the geometry description does not match the FEM mesh, this is also referred to as CutFEM.

In this thesis, we have contributed to a suitable algorithm for finding intersection points between the geometry description and the FEM mesh. We base our algorithm on Newton’s method to detect these intersection points. The results show that the intersection points are very accurate for regular triangles. The intersection points are found directly on the limit surface of Loop’s subdivision surfaces.
Acknowledgements

First of all I would like to thank my advisors, August Johannson (main advisor @ Simula Research Laboratory) and Michael S. Floater (UiO) for their helpful guidance and throughout this thesis. Especially my main advisor August who have provided me with a better understanding and more practical view of mathematical topics. I also want to thank my parents and brothers for their support during my studies, for which I am very grateful. Lastly but not least I want to thank the other students that I’ve met during my studies.
3.3.2 Axis-aligned bounding box .................................. 32
3.3.3 Object-oriented bounding box ................................. 32
3.3.4 Intersection testing ............................................. 32
3.4 Root Finding Methods ............................................. 34
  3.4.1 Bisection Method ............................................. 34
  3.4.2 Newton’s Method .............................................. 35
3.5 A Loop Subdivision Surface/Line Intersection Algorithm ... 36
  3.5.1 Implementation ............................................... 38

4 Results ........................................................................... 43
  4.1 Surface Results ...................................................... 43
  4.2 Newton’s Method Results ......................................... 44
    4.2.1 Radius Test Results ......................................... 44
    4.2.2 Time consumption comparison ............................. 45
    4.2.3 Convergence .................................................... 45

5 Conclusion and Future Work ........................................ 49
  5.1 Future Work .......................................................... 49
    5.1.1 Speedup using parallelism ................................... 49
    5.1.2 Extension to Quadrilateral Surfaces ...................... 50
    5.1.3 Improvements of the Subdivision Surface/Line Segment Algorithm .......................................................... 50
    5.1.4 Integration: Finite Element Method Solver ............. 50
    5.1.5 Integration: Existing 3D Software ......................... 50

Bibliography ...................................................................... 52

Appendices ........................................................................ 54
Chapter 1

Introduction

*Finite element method* (FEM) is a numerical method that has been around for some decades and used for approximating a solution for *partial differential equations* (PDEs) over a mesh. Within the field of computer graphics, subdivision surfaces have over the years become a highly used technique for efficient and detailed mesh rendering, especially in real-time rendering applications. This thesis will look at the possibility of making use of subdivision surfaces with finite element methods.

1.1 Motivation

To simulate a physical phenomenon on parts of a structure or the structure as a whole. Types of such phenomena that might occur could be stress, heat, and flow PDEs are used to represent these phenomena mathematically. However, since PDEs usually are solved on complex domains, a solution of them are therefore not possible to achieve using regular analytical methods. Meaning that the equations are impossible to solve by hand. Luckily for us, there exist methods that can be used to compute an approximation of the PDEs. One of them is FEM, which will be the focus of this study.

Looking deeper into the requirements for FEM to work, they consist of three major things; an equation (the PDE), some parameters from the real world that is needed by the PDE, and lastly a geometry description. A mesh is required to describe the geometry that the PDE is analysed over. Since many of the 3D models that we want to analyse a PDE over is within the field of engineering. These geometries are, often provided as *computer-aided design* (CAD) models.

[^distmesh]: http://persson.berkeley.edu/distmesh/
Since the geometry description provided to the FEM solver software is usually not a mesh, we often have to generate a new mesh that matches the geometry description. This process uses something called a mesh generator, such as Gmsh\footnote{http://gmsh.info} to generate these meshes. Mesh generators tend to make errors in the mesh if the model is not perfect. Errors that might occur is where the CAD model has a high level of detail (LOD), resulting in the generator generating too many polygons within specific areas. If such errors happen then the modeller of the CAD model needs to go back and redraw regions of the model that contains errors. Fixing errors in this matter could be, not only time-consuming but also leave out details from the real model that should be retained in the mesh.

1.2 Project description

This thesis has as purpose to investigate the possibility of using subdivision surfaces to provide a FEM solver with a geometry based on the original geometry description. The basic idea of FEM is to find an approximate solution for a PDE on a mesh. More often this mesh, which we can call the FEM mesh is often the same as the geometry mesh. When the FEM mesh and the geometry description do not match, we have a so called non-matching method. CutFEM is one example of such method.

We will use subdivision surfaces as geometry description for the CutFEM.

Contribution

The contribution of this thesis will mainly be that the process of generating a mesh that matches the geometry to describe the geometry used by FEM. Meaning that one does not need to use a mesh generator to generate a mesh. Leaving this process out and supply a way that uses the original control mesh would be time-saving for those who create and analyses a model.

1.3 Outline of the Thesis

- Chapter 2 Here we will be providing some background theory that is useful for the rest of the thesis. Here we provide a more detailed description of different topics within both FEM and computer graphics.
• **Chapter 3** In this chapter we will give a detailed explanation of some different algorithms and methods that we have investigated. As well as different tools and software that was used to create an implementation.

• **Chapter 4** We present our results from the implementation and various tests that we did in this chapter.

• **Chapter 5** This chapter concludes the thesis, providing a conclusion of the results from chapter 4. We also present some future work that can expand on this thesis.
(a) Example of FEM mesh and geometry description. The circle is the geometry description while the grid surrounding it is the FEM mesh. Blue squares in the figure indicates elements that are inside the geometry domain ($\Omega$) and yellow indicate elements on the boundary of the geometry description ($\partial\Omega$).

(b) Example where the FEM mesh matches the geometry mesh, created using `distmesh` in Matlab.
Chapter 2

Background

This chapter presents essential material for this thesis. We start by giving a broader explanation of FEM, and the widely used marching cubes algorithm. Then we move on to topics that concern subdivision surfaces which we start this section by explaining different surface descriptions, including subdivision surfaces. Lastly, we will give an explanation of the limit surface of a control mesh.

2.1 Finite Element Method

In this section we will dedicate more space for a briefly introduction of the finite element method (FEM), even though the thesis will be more concentrated around methods within the fields of computer graphics and computer science.

For describing laws of physics, we often do this by using partial differential equations (PDE). PDEs are often describing problems that are space- and time-dependent [1]. Since PDEs are often unsolvable using conventional analytic methods, instead FEM is used to approximate a solution for the PDE. Often, when using FEM the PDE is approximated over a complex domain.

FEM approximates a solution for the PDE over a provided geometric domain, and often this domain can be very complex. In FEM we use two different meshes, one of which is the mesh used to do the approximations over (the FEM mesh) and in addition we have a geometric description that is provided to the FEM solver.

A software program that solves these FEM problems is usually referred to as a FEM solver and the usage of such tools is referred to as finite element analysis (FEA). We want to approximate the PDE over the geometry description, meaning that for an optimal solution the two should match.
However, in most cases the geometry description and FEM mesh does not match completely as we shall see.

A major goal in using FEM is to compute

\[ \int_{\Omega} f(x)dx \tag{2.1} \]

, we will therefore use the notation

\[ \int_{\Omega} = \int_{\Omega} f(x)dx. \tag{2.2} \]

We will use the notation \( \{k\} \) to indicate a set of \( k \) elements of the FEM mesh used. There are conventionally two different cases of relations between the geometry description and the FEM mesh, that is solvable. Firstly we have the case where the geometry description matches the FEM mesh, written as

\[ \int_{\Omega} = \sum_{\{k\}} \int_{k}, \tag{2.3} \]

which implies that a correct solution for the PDE is achievable. An example for this case is a perfect cube as geometry description and smaller cubes that fit the geometry description as FEM mesh.

The first case is only achievable for very simple geometry descriptions, these geometry descriptions are thus almost never used in FEM since we would rather work with realistic models. We thus have a second case which is expressed as

\[ \int_{\Omega} \approx \sum_{\{k\}} \int_{k}. \tag{2.4} \]

In this case the FEM mesh is an approximation over the geometry description. It is in this case assumed that FEM mesh and geometry description fit in some way, however, the mesh is approximated around the boundary of the geometry description.

A third case is introduced in which the FEM mesh does not fit the geometry description. For this case, a method called CutFEM [4] is usually applied. We write this third case as

\[ \int_{\Omega} \neq \sum_{\{k\}} \int_{k}, \tag{2.5} \]

this case will be one of the main focus of this thesis. Here we want to find a FEM mesh so that approximating a solution over the geometry description is
possible, for doing this we must cut elements that are located on the boundary of the geometry description. We often refer to elements on the boundary as cut elements as the geometry description cuts the FEM mesh element. In order to be able to compute integrals over cut elements we need to use other techniques. A technique that could be used is called sub-triangulation, an algorithm for this is described in the following section Marching Cubes. Other algorithms exists and can be used for cutting the elements, such as the Monte Carlo algorithm. When using Monte Carlo algorithm to cut elements we make the assumption that the area with highest concentration in point markers is inside the geometry description. Meaning that an element should be cut between the area with high concentration and the area with low concentration of points.

### 2.2 Marching Cubes Algorithm

*Marching cubes* is an algorithm that creates triangular meshes from 3D related data. It was presented by William E. Lorensen and Harvey E. Cline in 1987 while they were working as researchers for General Electric Company (GE) [10]. It begun as an algorithm for obtaining better and also more detailed visualization of medical images; such as *computed tomography* (CT) and *magnetic resonance* (MR) scans. These types of scans produce many two dimensional images. By using the two-dimensional images taken by the scan, we can create a scalar-field. From which we can extract a three-dimensional mesh.

The purpose of the marching cubes algorithm is to generate a three-dimensional mesh from the scalar-field fast and in an efficient manner.

![Figure 2.1: The 15 possible base cases for marching cubes](image)
In figure 2.1 we see the fifteen different base cases of the marching cubes algorithm. However, from the base cases, we can form 256 unique combinations by "rotating" the corners in the base cases.

We notice the highlighted, orange coloured corners on the cubes in figure 2.1. These corners are located inside the mesh, while the other on the outside. The different combinations of the corners situated inside or outside of the mesh form a configuration. Such configuration needs 1-byte memory for storage, because of the maximum of 256 combinations that we have. This byte of memory contains 8-bits, in which each of these bits represents a corner in the cube. Corners issued outside of the mesh is represented in the byte by 0, and corners inside are set to 1.

E.g. if we look at the base case number three in figure 2.1, we can see that the two lower corners in front are inside the mesh. As a result, we store the base case as the following byte, 0000 0011.

A polygonal mesh is then "extracted" from these voxels based on their configuration.

Though marching cubes were initially intended for visualisation of medical images, it has more and more over the years the algorithm has been adapted for more and different fields of research. This includes CutFEM where it is used to find the surface and volume in elements that are cut by the geometry description.

2.3 Surface descriptions

We usually divide surfaces into two categories; parametric, and implicit surfaces. The difference between these two surfaces is how they are defined. Parametric surfaces is defined by a collection of control points. Whereas implicit surfaces are defined by a single function that describes the whole surface.

2.3.1 Parametric Surfaces

Parametric surfaces are defined and represented by a small number of control points [3, p. 592]. With these control points we can for example use triangulation to connect the control points into triangles which forms a representation of the surface.

There are advantages using parametric surfaces instead of implicit surfaces. The main two reasons is that, firstly one can animate the control points which is necessary for computer graphics. The second reason being
that; since the graphic processing unit (GPU) is very efficient for rendering triangles. The GPU uses a method called tessellation to transform control points into triangles on hardware level.

As we will see later, Bézier patches are one example of parametric surfaces that is used in this thesis. Another example of parametric surface is subdivision surfaces which are described in more details in chapter 2.4.

**Example: Computer-Aided Design**

*CADC* (CAD) is a technique used in the industry to make three dimensional models, mostly prototypes of different real-world objects. Non-uniform rational B-splines (NURBS) are commonly used in CAD to describe the geometry. NURBS provides great flexibility for modelling which makes it good for CAD modelling. It is widely used to describe geometries that is used in finite element analysis.

### 2.3.2 Implicit Surfaces

Implicit surfaces is a class of surfaces which can be useful as one does not need to describe each point on the surface explicit [3, p. 606], but instead uses implicit function to describe the surface.

An implicit function is given as

\[ f(x, y, z) = 0. \]  

(2.6)

E.g the implicit function for a sphere is given as

\[ f(x, y, z) = x^2 + y^2 + z^2 - r^2, \]  

(2.7)

where \( r \) is the radius of the sphere.

An advantage using implicit surfaces over parametric surfaces is that one can use logical operations such as AND and OR to extract different parts of the surface. This makes so called "inside/outside" testing easy since we only have to evaluate the implicit function. If \( f(x, y, z) < 0 \) then the point is inside the volume, if \( f(x, y, z) = 0 \) then the point is on the surface. Lastly if \( f(x, y, z) > 0 \) the point is located outside.

Partial derivatives of the implicit function gives the normal of the implicit function that describes an implicit surface given as

\[ \nabla f(x, y, z) = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right), \]  

(2.8)

where \( f \) is an implicit function.
2.4 Subdivision Surfaces

Subdivision has for a long time been a popular subject within the field of computer graphics. It is a powerful method for making a mesh smoother through a series of refinement steps, also called subdivision steps. By applying subdivision to a mesh, the polygons are divided into smaller polygons creating new vertices that helps smoothing the surface. Subdivision can be applied an infinite number of times. However eventually the surface will converge to a new surface where re-applying subdivision will result in a small, or no change at all in the surface. When reaching this point we say that the surface is the limit surface. There is a possibility to also compute the limit surface directly from the control surface. By doing this we would not need to use any subdivision steps between the control surface and the limit surface, saving a lot of computations.

Subdivision is especially popular in game development where objects in far view has none to few subdivision steps applied to them. As an object is getting closer to the camera, subdivision is applied an equivalent number of times. This makes the objects more detailed as they are getting closer, where the viewer would notice any flaws in the details. Using subdivision this way will save memory and computation power.

To get an overview of how subdivision works in practice, the main idea is to divide a large polygon into smaller polygons. However if we where just to create a new edge point on the middle of each edges we would end up with a flat and not more smoother surface than we started with. Therefore we have something called subdivision schemes. These subdivision schemes have some predefined rules on how existing vertices have their position updated and how new vertices are created. Each of the subdivision schemes are made for its own purpose, and thus have their own rules on how to update existing vertices and add new ones. By this we mean that it exists specific subdivision schemes for quadrilateral meshes and others for triangular meshes. Some subdivision schemes are approximating and others are interpolating. These subdivision schemes have although one time in common, namely that they all are generalizations of different types of splines.

Interpolating schemes are subdivision schemes where the control vertices and vertices that are added to the mesh during subdivision does not change. This means that the limit surface of the mesh is already defined as the surface would interpolate trough the control points. Another type of subdivision schemes are called approximating schemes, where the existing vertices have their position updated. Interpolating subdivision schemes might be preferred to use when one wants the surface to have the same shape as the control surface.
In the following section four subdivision schemes are described in more details. These four subdivision schemes have their own purpose. First, Loop’s subdivision scheme is described, which is an approximating subdivision scheme created for triangular meshes.

The basic transformation of a vertex over the course of subdivision steps are show as

\[ v^0 \rightarrow v^1 \rightarrow \ldots \rightarrow v^\infty, \] (2.9)

for an approximating subdivision scheme. Where \( v^0 \) represents a control vertex, and \( v^\infty \) represents the vertex on the limit surface.

Differences between triangular and quadrilateral meshes are mostly that quadrilateral meshes tend to have better continuity. This benefit means that textures are easier manageable and does not require that much manual work compared to triangular meshes. In movies where details are in high demand, making quadrilaterals perfect for this use. However, in today’s graphics processing units (GPU) are primarily optimised to use triangles when projecting meshes. Because of this optimisation towards triangles, they tend to be faster processed compared to quadrilaterals. Thus, making triangles the most used primitive for real-time rendering. However, quadrilaterals could be more detailed since triangles will have trouble around irregular vertices, due to the continuity around these vertices.

Continuity

When we look at different subdivision schemes, we also would like to know something about the quality of the subdivision scheme. Or more precisely how smooth the surface becomes after the subdivision scheme has been applied to the mesh. Continuity of the subdivision scheme helps us with this. It indicates the stability of the splines. We use \( C \) to denote continuity. That a subdivision scheme has continuity of \( C^1 \) means that the subdivision is generates a smooth surface and is stable for the 1st derivative of the splines. A \( C^2 \) continuity means that the subdivision scheme is stable for the 2nd derivatives.

2.4.1 Subdivision Schemes

The following section briefly describes the history and function of four quite different subdivision schemes. Two of which is approximating and two that are interpolating. Under these categories there is one subdivision scheme for quadrilateral meshes and one that supports triangular meshes. The first
subdivision scheme that is described is widely used for triangular meshes, while the second is widely used for quadrilaterals.

2.4.2 Catmull-Clark Subdivision Scheme

In 1987 Edwin Catmull and Josh Clark published their paper "Recursively generated B-spline surfaces on arbitrary topological meshes" [5] where they presented their research of what has become to known as the Catmull-Clark subdivision scheme. This is an approximating subdivision scheme that generalizes bi-cubic B-splines. In the refinement process, quadrilaterals are subdivided into four new quadrilaterals. The subdivision scheme can also handle meshes containing other primitives, such as triangles. If the control mesh where to contain triangles, then these are transformed to quadrilaterals after the first subdivision step.

Catmull-Clark subdivision scheme has been frequently used by Pixar in their movies, starting with their well known short movie *Geri’s Game* (1997). In the movie, subdivision was applied to Geri’s head, arms and clothing to make the character more life-like [8]. Pixar subsequently replaced their former standard technique using NURBS, because of the level of details (LOD) that subdivision provides near the limit surface.

For Catmull-Clark subdivision we say that a vertex is ordinary if it has a valence of 4, otherwise it is assumed to be extraordinary. Compared to Loop’s subdivision scheme where triangles with valence 6 is considered ordinary.

In figure 2.2 we can see the various masks that Catmull-Clark subdivision uses. We can see that the scheme generates two new points, one in the middle of each face and one in the middle of each quadrilateral. Then lastly we have the vertices that already existed in the mesh that have their position updated with a vertex mask.

The algorithm used for Catmull-Clark subdivision is pretty straightforward. First we need to generate the new face vertices using the equation. Then we compute the new edge vertices, and lastly we update the position of the existing vertices. This process is then repeated for each subdivision step.

- **Face mask** is used to create a new face vertex. A face vertex is a vertex that is centered in a face from the previous subdivision step. By looking at 2.2a we clearly see that it is calculated averaging the corner vertices of the face.

So if we denote the $v$ as the corners of the face a new face point is computed as
\[ f^{k+1} = \frac{1}{4} (v_0^k + v_1^k + v_2^k + v_3^k) . \] (2.10)

- **Edge point mask** is used when creating new edge points. We start off by finding the two faces that shares the edge that the new edge point should be located on. From the figure [2.2b] we see that the outer corners have a weight of \( \frac{1}{16} \) and the edge end points on the edge to the new edge point has weight of \( \frac{3}{8} \).

\[ e^{k+1} = \frac{1}{16} (p_0^k + p_1^k + p_4^k + p_5^k) + \frac{3}{8} (p_2^k + p_3^k) \] (2.11)

A newer method for computing new edge points is described in Real-time rendering[3, p. 623]. Where instead of using the outer vertices we can use the newly created face vertices. As well as the edge points on the edge of the new edge point and take the average position of these.

- **Vertex mask** is applied to existing vertices to update their position on the new surface. As we can see in figure [2.2c] it requires some more vertices compared to the edge and face masks. We now use the four surrounding faces that is generated by the new edge and face points.

\[ v^{k+1} = \frac{n-2}{n} v^k + \frac{1}{n^2} \sum_{j=0}^{n-1} e_j^k + \frac{1}{n^2} \sum_{j=0}^{n-1} f_j^{k+1} \] (2.12)

### 2.4.3 Loop’s Subdivision Scheme

Charles Loop presented in his master thesis (1987) a new subdivision scheme for triangular meshes, which became to known as Loop’s subdivision scheme [9]. Loop’s subdivision scheme was the first subdivision scheme that, it is explicitly for triangular meshes [3]. Loop’s subdivision scheme is an approximating subdivision scheme that handles triangular meshes. Loop’s subdivision scheme is based on quartic box-splines.

In figure [2.3] we can see a icosahedron being subdivided using Loop’s subdivision surfaces. Figure [2.3] shows the process from the control surface subdivided two times.

[https://commons.wikimedia.org/wiki/File:Loop_Subdivision_Icosahedron.svg](https://commons.wikimedia.org/wiki/File:Loop_Subdivision_Icosahedron.svg)
During the subdivision process, existing vertices have their positions updated and a new vertex added to each edge. Which leads to four new generated triangles for each existing triangles; meaning that there is $4^n$ triangles on the mesh after $n$ subdivision steps.

In his thesis Loop proved that Loop’s subdivision scheme has a $C^2$ continuity for regular vertices and a $C^1$ continuity for irregular vertices. Vertices with a valence of 6 are considered regular. We will only have extraordinary vertices that were extraordinary in the control mesh because the new vertices added to the mesh will always have a valance of 6 and be considered regular.

For updating an existing vertex from $v^k$ to $v^{k+1}$ we use the formula

$$p^{k+1} = (1 - n\beta)p^k + \beta(p^k_0 + \cdots + p^k_{n-1}),$$  (2.13)

and for generating a new vertex

$$p^k_i = \frac{3p^k_i + 3p^k_{i-1} + p^k_{i+1} + p^k_{i-1} + p^k_{i+1}}{8}, i = 0 \ldots n - 1 \mod n. \quad (2.14)$$

We also have $\beta$ which is a function of $n$ that is given by

$$\beta(n) = \frac{1}{n} \left( \frac{5}{8} - \frac{3 + 2 \cos \left( \frac{2\pi n}{n} \right)^2}{64} \right),$$  (2.15)

or as

$$\beta(n) = \frac{3}{n(n + 2)}. \quad (2.16)$$

The last variant of $\beta$, presented by Warren and Weimer [3, p 615], avoids the use of the trigonometric function cos which can be tedious for a computer to calculate. We can also see that using the last variant of $\beta$, the total number of calculations that are necessary is reduced compared to the original.

Later we will give this subdivision scheme a closer look, as we look on how we can evaluate the limit surface of this subdivision scheme.

### 2.4.4 Deng-Ma Subdivision Scheme

An interpolating subdivision scheme for quadrilateral meshes was in 2013 proposed by Chongyang Deng and Weiyin Ma[7]. Even though the subdivision scheme is relatively new, it is based upon a subdivision scheme that was proposed in 1989 by Deslauriers and Dubuc, called 2n-point subdivision scheme.

Deng-Ma subdivision scheme does subdivide the edges and the faces. The subdivision scheme defines rules for creating new edge vertices and face
vertices. Since the Deng-Ma subdivision scheme is interpolating, we do not update already existing vertices. We refer to the paper by Deng-Ma[7] for a detailed explanation of the rules for creating new vertices, as this is out of the scope of this thesis.

**Edge vertices rules**

From the paper, we notice that the subdivision scheme categorises extraordi-
nary vertices into two categories. The categories being edges whose valence is equal to 2, meaning that it is a corner vertex, and the other being an edge vertex with a typical valence of 3. In the case of an extraordinary vertex, we generate a new temporarily vertex. The temporarily generated vertex makes the vertex ordinary.

**Face vertices rules**

To generate new face vertices we need the newly created edge vertices. From these edge vertices, we can create new edges in two directions, we then see that we can generate two edge points which will be near centred in the face. The two new vertices are average, resulting in one vertex which is the new face vertex.

**Smoothness**

Deng and Ma did smoothness analysis of the proposed subdivision scheme where they showed that it has $C^1$ continuity for extraordinary vertices. But mentions that the continuity can be $C^2$ for extraordinary vertices where the valence $N = 5$, but never if $N > 5$[7].

### 2.4.5 Modified Butterfly Subdivision Scheme

In this section we will give a brief description of another subdivision scheme for triangular meshes. Modified butterfly subdivision scheme is a modification of the butterfly scheme, that was presented by Zorin et al [3, p. 616]. It is an interpolating subdivision scheme, compared to Loop’s subdivision scheme which is approximating. Meaning that we don’t update the already existing vertices, but only generates new ones. The subdivision schemes have different cases that describes how a vertex is created. These cases depends on things such as boundary edges, and extraordinary vertices. There are four cases that is considered.

- **Case 1: Ordinary** When both edge vertices are regular
• **Case 2: Semiordinary** When one edge vertex is ordinary and the other vertex is extraordinary.

• **Case 3: Extraordinary** When both edge vertices are extraordinary.

• **Case 4: Boundary** When the edge where we want to generate a new vertex is a boundary edge.

This subdivision scheme can create some bad surfaces if the control mesh is very complex. It is also more computation expensive than Loop’s subdivision, due to the number of vertices needed to generate a new vertex.

### 2.5 Limit Surface

Whenever we apply a subdivision scheme on a surface, it will get divided into smaller pieces which lead to a smoother surface. There comes the point when the changes in the surface are barely notable. We say that this happens after an infinite number of subdivision steps. The resulting surface then is called the **limit surface**, or also **exact surface** of the control mesh. We usually say that the control surface converges towards this limit surface.

Based on the type of subdivision scheme that we use this limit surface could be entirely different. If we were to use an interpolating subdivision scheme, which would maintain all the control points, it would take the surface longer (more subdivision steps) to converge toward the limit surface. On the other hand, there is approximating subdivision schemes that do not maintain the control points, and thus, the surface will get smoother with fewer subdivision steps compared to an interpolating subdivision scheme.

However, since approximating subdivision schemes does not maintain the control points the whole surface will change. For this reason, surfaces using approximating schemes tend to become smaller when a subdivision scheme is applied. Which means however that the surface is guaranteed located inside the **convex hull** of the control mesh. Interpolating subdivision schemes, on the other hand, is concentrated around the control points of the mesh so that the limit surface have more or less the same size as the control mesh.

#### 2.5.1 Limit Surface in Loop’s Subdivision Scheme

Here we will give a more detailed description of the limit surface, specifically the limit surface for Loop’s subdivision scheme. For this subdivision scheme we can find the position of a control point on the limit surface by simply changing the $\beta$. 

20
Since Loop’s subdivision scheme is an approximating subdivision scheme, the surface will shrink and not maintain its original form. There are methods that can control the shrinking of a surface to a certain point.
(a) Face mask.  
(b) Edge point mask.  
(c) Vertex mask.

Figure 2.2: Catmull-Clark subdivision masks.
Control mesh:

Subdivision step 1:

Subdivision step 2:

Figure 2.3: Example of Loop’s subdivision surface.

Figure 2.4: Example of Loop’s subdivision on icosahedron.
(a) New vertex masks.  
(b) Existing vertex masks.

Figure 2.5: Loop subdivision scheme masks.
Chapter 3
Methods and Tools

In this chapter, we present a suggested implementation along with some essential algorithms and methods within numerical approximation and computer graphics that we used in our implementation. In the first section **Tools and frameworks** a briefly description of the different tools and frameworks that is used in the presented implementation.

The first sections describe some important theory about the algorithms that are used in our implementation, while the section Setup contains information about the setup around the suggested implementation. We will present multiple methods and briefly describe why we did and did not choose the following methods and algorithms to be a part of the implementation.

3.1 Tools and frameworks

In this section a more detailed report on what techniques, data structures and algorithms that we used in the implementation. The suggested implementation was written in the programming language C++ and uses Eigen linear algebra library for the most matrix calculations. Eigen has implementations for SIMD (single input multiple data) vectorization in their library so it is easy to optimize the implementation. For visualization we used OpenGL accompanied by GLEW and GLUT.

3.1.1 Programming language

We chose to use the programming language C++ for the implementation because of the speed and flexibility of the language. C++ language is a low-level programming language that was developed by Bjarne Stroustrup in [http://en.cppreference.com/w/](http://en.cppreference.com/w/)
the early 80s. Despite the age of the language, it is still considered modern due to various updates over the years. C++ is very similar to C, as it lets the developers work with memory management, with the exception that it is object-oriented (OOP).

For different test result visualisation, mostly in the form of graphs, we used python and matplotlib for simplicity.

**Single vs. Double Precision**

In respect to precision in C++ we have created some typedefs based on the preferred precision. Since Eigen matrices and vectors are templates based on precision they are type defined to `RealMatrixX`, `RealMatrix2`, `RealMatrix3`, and `RealMatrix4`. For the basic type `float` or `double` we have the type definition `real`. We test our implementation using double precision. However the differences might not be too clear using double precision over single precision, in these cases one may want to use single precision instead. Type definitions in C++ provides a neat way to switch between the two.

### 3.1.2 Eigen

Eigen\(^2\) is a high performance linear algebra library for C++. The library is also optimised by using single input multiple data (SIMD) vectorization. This makes the job of optimising the implementation much easier since most of our calculations is using the Eigen library. We use Eigen widely in the implementation for vectors, matrices and linear systems.

### 3.1.3 Computer Graphics Libraries

Our implementation uses a popular stack for computer graphics. Namely the OpenGL/GLUT/GLEW stack. These libraries are cross-platform available, meaning that they will work on all the major operating systems, Windows, OS X, and UNIX.

- OpenGL\(^3\) (Open Graphics Library) is an open-source application programming interface (API) is supported on most graphics cards and operating systems.

\(^2\)http://eigen.tuxfamily.org/index.php?title=Main_Page
\(^3\)https://www.opengl.org
- GLUT\textsuperscript{4}(OpenGL Utility Toolkit) is used for window creation for OpenGL. The library is easy to use and does not require platform specific implementations as it is cross-platform.

- GLEW\textsuperscript{5}(OpenGL Extension Wrangler) provides various functions to check for supported OpenGL features on the current running system.

The implementation is built using CMake and tests are written using Google Test. Google Test is developed and maintained by Google and is used to do unit testing for individual functions. Unit testing is particular good in our case when we are checking a problem where we know the solution instead of manually checking tens or hundreds of intersection points the tests will do this work for us. Reasons why we chose this combination of tools is heavily linked with the speed and power that the C++ programming language provides and that it is more or less cross platform. It is also one of the most common programming language of use when it comes to real-time rendering, because of its speed and power. We chose C++ over the old C language because of the object-oriented programming (OOP) paradigm that C++ provides. Loop’s subdivision scheme was chosen as our primarily subdivision scheme, as explained earlier. This means that the implementation is primarily targeted for triangular meshes. However it should not be hard to develop support for quadrilateral meshes. Using Catmull-Clark subdivision scheme or a similar subdivision scheme instead or in addition to the current implementation with Loop’s subdivision scheme. The first technique presented is based on Stam’s paper\textsuperscript{12}. It is used for local evaluation of a subdivision surface using something called Bezier patches. We evaluate each triangle that has an intersection in the control mesh so that we can find the intersection point on the limit surface.

3.2 Algorithms

Here we present some core algorithms that was used in our implementation in more details. The first algorithms concerns the setup of meshes, while the last algorithms mostly concerns the mesh grid.

3.2.1 Local evaluation of Subdivision Surfaces

In our solution, we use basis functions for the Newton-Raphson calculations. These basis functions are based on the same basis functions Jos Stam uses in

\textsuperscript{4}https://www.opengl.org/resources/libraries/glut/
\textsuperscript{5}http://glew.sourceforge.net
his paper [12]. He bases his basis functions on a technique for converting box splines to triangular Bezier patches; this means that our Bezier patches need to be identical to his to make the basis functions applicable to our solution.

Each of the Bezier patches that we evaluate will consist of $K = N + 6$ vertices. When we subdivide the patch, it will consist of $M = K + 12$ triangles. We let $C$ denote a $K \times 3$ matrix containing the position of the $K$ control vertices. We will also use a matrix that contains the weight rules for the patch, denoted $\tilde{A}$. When we multiply these two matrices we will get the next subdivision step of the Bezier patch, so

\[ \tilde{C}_{t+1} = C_t \tilde{A}, \]

where $\tilde{C}_{t+1}$ denotes the next subdivision step.

This method of subdivision comes in handy since the basis functions that we want to use in our calculations is only applicable for triangles with regular valence. If we have a triangle where one vertex has valence other than six or three we have to subdivide the patch another step to evaluate the patch. When we subdivide an extraordinary triangle we will get four new triangles located inside the original triangle. From this, we can evaluate three of the four triangles, which are ordinary. The one triangle in the patch that has the original extraordinary vertex as one of its vertices needs to be subdivided again. After the first subdivision step, we are therefore able to evaluate $\frac{3}{4}$ of the triangle.

Bezier patches

As mentioned earlier we will generate Bézier patches for each triangle that we want to evaluate. Some literature defines the Bézier patches as a patch that consists of quadrilaterals and Bézier triangles to be somewhat similar but for triangles instead [3]. We will see that our Bézier patches, containing triangles are most similar to Bézier patches, due to the centroid triangle evaluated, and marked with orange colour in figure 3.1. Were also the triangles marked with white colour is the centroid triangle’s immediate neighbours. We also notice that the indexing of the Bézier patch is crucial, due to the basis functions that we are going to apply.

3.2.2 Half-edge Data Structure

Half-edge data structure, also known as doubly connected edge list (DCEL) is a well-known data structure used for mesh triangulation. It provides a fast and efficient way to traverse edges on the mesh. Half-edge data structure creates what is called a twin edge for each edge. Twin edges have the opposite
(a) An ordinary Bezier patch.  

(b) An extraordinary Bezier patch.

Figure 3.1: Indexing of Bezier patches.

direction as the original edge. Since each edge has an opposite twin-edge, the structure is considered fast to traverse since we can easily retrieve the neighbour triangles. So when we are to retrieve the Bezier patch of a triangle, we can traverse its edges to get the neighbour triangles that make the Bezier patch.

Table 3.1: Half-edge properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Size (bytes)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td>8</td>
<td>Pointer to the source node of the half-edge</td>
</tr>
<tr>
<td>destination</td>
<td>8</td>
<td>Pointer to the destination node of the half-edge</td>
</tr>
<tr>
<td>next</td>
<td>8</td>
<td>Pointer to the next edge</td>
</tr>
<tr>
<td>prev</td>
<td>8</td>
<td>Pointer to the previous edge</td>
</tr>
<tr>
<td>twin</td>
<td>8</td>
<td>Pointer to the twin edge</td>
</tr>
<tr>
<td>triangle</td>
<td>8</td>
<td>Pointer to the face that the edge is connected to</td>
</tr>
</tbody>
</table>

Total size 46

From the table 3.1 we can see that the required memory is 46 bytes for each half-edge, this adds up to a total of 138 bytes per triangle. In some cases where low memory usage is crucial one might retrieve some of the data during runtime rather than storing the data. The source and destination
nodes could also be stored in a sorted array. To find the twin edge, we need to compare the source and destination node of two half-edges \((e_1 \text{ and } e_2)\). In the case of using source and destination nodes rather than a sorted array we would compare the source node of \(e_1\) with the destination node of \(e_2\), and likewise on the destination for \(e_1\) and source node of \(e_2\). If we instead where to use a sorted array, in which the vertices are sorted after the \(x\), \(y\), and \(z\) coordinates. The approach is to compare the two arrays and check if they are equal. Sorting the vertices are achieved merely by first compare the \(x\) coordinates, if they are equal then we compare the next coordinate \(y\). Again if the \(y\) coordinates are identical, we move on to the last coordinate (working in three-dimensional space) \(z\). The vertices are equal to each other if all the coordinates are identical. However, this should not happen for half-edges. If the vertices are equal, meaning they lay on top of each other, we may assume something is wrong with the triangulation or the geometry itself. An half-edge who have no twin edge is considered a boundary edge of the mesh, also meaning that the surface of the mesh is non-manifold. We say that a mesh is non-manifold if the geometry could not have existed in the real world. Therefore at most times when we simulate real-world problems, we would prefer to use a manifold surface. Eventually when all edges in the half-edge structure are connected with its twin-edge and a polygon we would end up with an adjacency graph [3, p. 543].

As we can see in figure 3.2 the half-edges of e.g the triangles \(a\) and \(b\) are moving in the opposite direction.

### 3.2.3 FEM mesh

Using the finite element method require us to be aware of which elements that are cut by the geometry description. These elements are constructed as
a three dimensional grid. In our implementation the grid is constructed as a collection of grid elements, almost like axis-aligned bounding boxes, with the exception that we also generate and store each edges of the grid element. We remind the reader that we use the FEM mesh to solve a PDE over the geometry description. In each of the cut elements we need to find the volume and surface to compute the integral on the element.

3.3 Collision Detection

Collision detection is the problem in which one wants to check if two objects collide. Collision detection, usually done by checking if there exists an intersection or overlap between the two objects. When solving a collision detection problem, we like to make it as simple as possible to save computational power. When working with collision detection in computer graphics, we may need to do hundreds or thousands of such collision tests between objects. Therefore, a mesh or a portion of the mesh enclosed within a primitive referred to as bounding volumes. These different bounding volumes described in more details below as well as some known algorithms that we use in our implementation.

3.3.1 Bounding Volumes

We usually want to do collision detection between an object \( S \) to test if it collides with a set of other objects, we denote this as \( T = T_1, T_2, \ldots, T_n \) that we want to test whether object \( S \) collides into \( T \). Since \( n \) could, and usually is large and the complexity of computing and testing for collision. A normal approach to the problem is to minimise the problems, which in this case we can minimise the surface of the objects that we do collision testing between.

To minimise the surface of an object we use so-called bounding volumes. These bounding volumes are usually simple objects that are covering the original object, or mesh that we test. Even though bounding volumes have different shapes, the regular way to test whether they collide or not is to check if the two bounding volumes overlap. However, collision testing using bounding volumes is fast and memory efficient, but since we are simplifying the surface, it can also mean that the two objects within the bounding volumes do not collide.

There exist many types of bounding volumes, such as spheres, capsules, and boxes. Mainly there are two types of box bounding volumes. These are axis-aligned bounding box and object-oriented bounding box. We describe these bounding boxes in more details later.
Another essential component for doing collision detection is the algorithms that we use. Many such algorithms do exist. The algorithms work for the different combination of bounding volumes and, or other primitives, such as rays. We will discuss two of these algorithms that we will use in the implementation.

### 3.3.2 Axis-aligned bounding box

*Axis-aligned bounding boxes* (AABB) is a type of bounding volume that is capable of fast computing and a good prediction of collision detection. AABB is never rotated which means that the algorithms used for collision testing on an AABB and another primitive does not have to take angles into account, which makes collision detection, for example between two AABBs very simple and efficient.

An AABB, usually defined by two points, namely the maximum and minimum of the AABB. From the minimum and maximum points, we can test for collisions.

### 3.3.3 Object-oriented bounding box

*Object-oriented bounding boxes* (OOBB) is very similar to an AABB with the difference in that it can be rotated. This property makes it a little bit harder to test for collision. Because of this most common way to use this type of bounding volumes is on rotated objects and AABBs on other objects.

In this thesis, we are only considering AABBs as we will be testing the implementation using a grid where none of the objects is rotated. However, we could have used it on the triangles, but because of the complexity of OOBBs and the number of triangles we are testing against, it would be overall faster using AABB.

### 3.3.4 Intersection testing

Following is a description for each of the intersection tests that we use in the implementation. We use this intersection tests to minimise the number of evaluation of the limit surface we have to do. Even though we intend to do intersection testing on the limit surface, we do not explain it here but later on in chapter 3.
AABB/AABB Intersection Test

Testing two axis-aligned bounding boxes whether they overlap each other or is a simple if statement where the returned value is simply true or false. These boolean checks are given as

\[
f(A, B) = (A_{x_{\text{min}}} \leq B_{x_{\text{max}}} \land A_{x_{\text{max}}} \geq B_{x_{\text{min}}}) \land \\
(A_{y_{\text{min}}} \leq B_{y_{\text{max}}} \land A_{y_{\text{max}}} \geq B_{y_{\text{min}}}) \land \\
(A_{z_{\text{min}}} \leq B_{z_{\text{max}}} \land A_{z_{\text{max}}} \geq B_{z_{\text{min}}}),
\]

where \( A \) and \( B \) are two independent axis-aligned bounding boxes.

The formula above indicates that we test for overlapping. First, we check for overlap on the \( x \)-axis, then for the \( y \)-axis, and lastly the \( z \)-axis is tested. We need to check all the axes for intersection; this is because one box could be placed a little on the side of the other. Which could result in one of the axis tests could return true, but the others would have been false. Therefore we assure that a part of one box is located inside the other, like in the figure below (3.3).

![Figure 3.3: Example of two AABBs intersecting.](image)
**Ray/Triangle Intersection Test**

A ray has a start point in space moving in a given direction over an infinite distance. The definition of a ray \( R(t) \) is given as

\[
R(t) = O + tD,
\]

where \( O \) is the origin or start point of the ray, and \( D \) is the direction (normalized) of the ray.

A ray could also be looked at as a line without an endpoint. However, in our implementation, we use the same algorithm for grid element edges as for ray-triangle intersection testing. Because the ray does not have an endpoint, we would have to check the resulting point to see if its location is on the grid element edge between its corners.

An efficient and well-known algorithm for doing such ray-triangle intersection testing is the Möller–Trumbore ray-triangle intersection algorithm \[11\]. The algorithm is known to be fast and efficient as the algorithm will cancel and return as soon as possible if there is a single possibility that the ray and triangle do not intersect. By using this algorithm, the triangle is translated into Barycentric coordinates, and by doing this, it bypasses the need of generating planes to test the triangle with. Leading to less memory consumption for each test.

### 3.4 Root Finding Methods

There exist many methods for root finding for a continuous function \( f \). However, since these methods are approximating the root, in some cases we can see that even if a root exists it may not be picked up by the method. Each of them comes with some pros and cons, we have to choose between things like speed and accuracy. In the following section two well known methods for root finding is described. And in the end a justification for why we chose the method that we did.

#### 3.4.1 Bisection Method

An example of a root finding method is the *bisection method*. By bisect, we mean that a set (in this case a set of real numbers \( \mathbb{R} \)) that is divided into two equal sized smaller parts of the interval. This method is more reliable than Newton’s method, as it guarantees convergence towards the roots. However, this comes at the price of speed. The way it works is that one bisects an interval, in which the root of the function is located at some position. This process is then repeated in a nearly infinite number of recursion steps.
We start with an interval \([a_0, b_0]\) and calculate the midpoint of the interval as
\[
c = \frac{a_0 + b_0}{2}.
\] (3.4)

The interval thus divided by the midpoint \(c\). To decide if the root is fair enough we supply the method with a tolerance usually by using \(a_0\) as a reference. Reason for this is, so we don’t go out of the initial interval. We could also check that the initial interval contains root(s) by evaluating \(f(a)\) and \(f(b)\); to then see if they have different sign. However, it could be that the function moves from negative to negative again within the interval.

We chose not to use the bisection method mainly because of the speed and amount of calculations required; instead, we decided to use Newton’s method. Bisection method might be a good alternative when the derivatives are unknown. Since the derivatives of Bezier patches are already know to us, Newton’s method seems like a better choice for a root-finding method.

We begin with what we call a start interval. Which then is divided into two new intervals that have equal length. Next, we see in which of the two intervals that the midpoint appears and use that interval for further calculations.

Then the midpoint of the interval is calculated, this is the interval divider. So now we have two intervals that we can use in the trailing step, we need to choose the interval in which the root is located. This steps are then repeated until we evaluate the function \(f(c) = 0\).

This method implies that one should have some knowledge in advance that the root is located somewhere on the initial interval so that it does not iterate for an infinite number of times.

### 3.4.2 Newton’s Method

Another well-known root finding method is called Newton’s method or Newton-Raphson method. Like the Bisection method described earlier (3.4.1), Newton’s method requires a function \(f\) to be evaluated. Unlike the Bisection method it requires in addition the evaluation of the derivative of \(f\).

The basic equation for Newton’s method is given by
\[
x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},
\] (3.5)

where \(x_{n+1}\) is the next iteration of the method, and \(x_n\) the current iteration. The proof of Newton’s method can be read from theorem 6.3.1 [6, p638].

Newton’s method is considered fast since the convergence rate is relatively high, meaning that the method requires fewer iterations to converge towards
a tolerable result. In addition to being fast the method is also very accurate in the approximation of roots. In one-dimension the method will return where \( f \) crosses the \( x \)-axis. If the function crosses the axis multiple times, the result will depend on the start guess \( (x_0) \) we provide.

The method also depends on a good start guess as a bad choice could result in wrong intersection point if the function intersects multiple times. Or it might not find any roots at all if the start guess is bad.

We can express the linearization of Newton’s method around 0 in two variables as

\[
f(v, w) \approx f(v_0, w_0) + \nabla f(v_0, w_0)(v - v_0, w - w_0),
\]

The same function can also be expressed on the matrix form as

\[
\begin{bmatrix}
v \\
w
\end{bmatrix} \approx \begin{bmatrix}
v_0 \\
w_0
\end{bmatrix} - \nabla f(v_0, w_0)^{-1}f(v_0, w_0),
\]

where \( \nabla f(v_0, w_0)^{-1} \) is the equivalent to the Jacobian matrix \( (J) \)

### 3.5 A Loop Subdivision Surface/Line Intersection Algorithm

With FEM one usually works with three-dimensional meshes, this means that an intersection algorithm should work for such meshes. In this section, we present an efficient algorithm for intersection testing on three-dimensional data. More specific, between limit Loop’s subdivision surface and a line segment. The algorithm uses a multivariate version of Newton’s method combined with Bezier patches of the surface.

The multivariate version of Newton’s method that we show here uses two inputs and outputs, thus \( f : \mathbb{R}^2 \to \mathbb{R}^2 \). We will use two variables \( v, w \in [0, 1] \), in addition the basis functions uses a parameter \( u \) which is defined as \( u = 1 - v - w \). In order to use the two variables \( v \) and \( w \) instead of the commonly used Euclidean space coordinates \( x, y, z \), we need to project the triangle down to two-dimensions. We does not have to supply a third parameter \( u \) since this is origo in the Cartesian coordinate system. Each of the parameter represents a vertex in the triangle, this triangle is placed so that \( u \) is located at origo an

Usually in the Cartesian coordinate system, one works with the coordinates \( (u, v) \). We let each coordinate represent vertices of the triangle. \( u \) represents the corner that is placed in origo of the coordinate system i.e \( (0, 0) \). Then we have \( w \) representing the \( v \)-axis, and \( v \) the \( u \)-axis. So basically we
switch the \((u, v)\) coordinates with \((v, w)\). The unit triangle is showed in figure 3.4 where we can also see how a line segment intersecting the triangle would appear. The line segment that intersect with the triangle would appear as a point. The depth of the intersection point is therefore unknown until we map the triangle back to three-dimensional space.

![Unit triangle](image)

(a) The unit triangle.  
(b) Unit triangle with an intersection in the orange point.

Figure 3.4: The unit triangle in \(vw\)-coordinates.

We project the triangle down to two-dimensions by taking the orthogonal coordinates from each corner of the triangle; the orthogonal coordinates are based on the origin of the line segment. By taking the orthogonal coordinates, we mean that we take the coordinates along the axes in which the line segment does not move along. So if the line segment does move along the \(x\)-axis, the direction would be a vector on the form \((1, 0, 0)\) as we use unit vectors to represent the direction. We would then extract the \(y\) and \(z\) coordinate from position vectors since the line is moving along the \(x\)-axis. After we have removed the coordinate corresponding to the direction, we get a ”new” position vector becomes \((y, z)\).

For to why we choose to use orthogonal coordinates in this manner, is the fact the line segment becomes a single point if we view the triangle as transparent in two-dimensional with the line going through it and look directly at the line segment, we would in fact only see the start point of this line segment. However, we need to adjust the depth of the intersection point later, which we achieve by using some defined basis functions that was presented by Jos Stam [12] and included in the appendix for more straightforward reading.
From Stam’s paper [12] we have that
\[ s(v, w) = C^T b(v, w), \]  
(3.8)
where \( s \) is the surface being evaluated for which we have the mapping \( s : \mathbb{R}^2 \to \mathbb{R}^3 \). \( C^T \) is the coordinates of all vertices from the Bezier patch, and \( b \) is the basis function. By introducing \( r \) as the ray function representing the line we provide to the method, we can see that the

By generalising our problem into
\[ s(v, w) = r(v, w) \]  
(3.9)
, where \( r \) is the origin of a line segment, we can see that a intersection point/root is found when \( s(v, w) - r(v, w) \approx 0 \). Out from this observation we can define \( f \) as
\[ f(v, w) = s(v, w) - r(v, w). \]  
(3.10)
An optimal solution is found when \( f(v, w) = 0 \), as we introduce an error tolerance (\( \epsilon \)), a root could be found on the interval \([0, \epsilon]\), where 0 indicates an exact root/intersection with no error.

\subsection*{3.5.1 Implementation}

In our implementation we will use this two variable Newton’s method. We decided that this method was one of the best choice since we are familiar with the basis functions of a Bezier patch from Jos Stam’s paper [12]. And from these basis functions we can compute the derivatives, making Newton’s method an excellent choice for root finding. Stam’s paper provides twelve basis functions that we use in our solution, where each of the basis functions represents one vertex on a regular Bezier patch. Since we already know these basis functions, we have the ability to calculate derivatives based on these basis functions. And then use the derivatives in our implementation of the Newton’s method.

We will also, in a combination with Newton’s method organize the mesh in such manner that we can easily extract something called Bezier patch [12] for the triangle being evaluated. Use of the word Bezier patch might be a little misguided here since, some literature define it as a patch of quadrilaterals with a center quadrilateral. And for triangles we have Bezier triangles which does not have any center triangle. In this thesis the word Bezier patch will be used as in [12], meaning that we have a centroid triangle (the triangle that we want to analyze) and all neighbours of this triangle. We can also say that it is a Bezier patch defined for triangular meshes, which we will think of it in this thesis.
Input: $p = \text{BezierPatch}$, $b(v, w)$, $vw = (0.125, 0.125)$

Data: $err = 1, \epsilon = 10^{-8}, it_{max} = 100$

Result: Intersection point between an edge and surface

for $i \leftarrow 1$ to $it_{max}$ & $err > \epsilon$
do
  $J = \begin{bmatrix} p^T \frac{\partial b}{\partial v}, p^T \frac{\partial b}{\partial w} \end{bmatrix}$
  $f = p^T b - l_{start}$
  $vw = vw - \text{SolveColPlvHouseHolderQr}(J, f)$
  $err = \|f\|$
end

return $p^T b$

Algorithm 1: Newton intersection algorithm

The basic outline of Newton’s method part of the algorithm is showed in pseudocode\[1\]. We can see that the algorithm has three parameters. First we have the parameter $p$ which is the Bezier patch that is associated with the triangle that we evaluate, this is an $12 \times 3$ matrix. Each row in the matrix $p$ corresponds to a vertex in the Bezier patch, and the columns corresponds to $x, y, z$ coordinates respectively. Secondly, we have a function $b(v, w)$ as a parameter, this parameter is basically a $12 \times 1$ matrix that contains 12 basis functions, one for each vertex in the Bezier patch. In addition to the $b(v, w)$ parameter we will need the basis functions derivatives with respect to $v$ and $w$, which we have pre-computed.

The last parameter $vw$ is the initial guess of root that we supply to Newton’s method, this is a $2 \times 1$ matrix that contains a guess along each axis. The guesses can be randomized to a certain degree. Since we use a triangle that is perpendicular in origo and spans from $v = 1$ and $w = 1$, we see that half of the space within the coordinate system will be outside of the triangle. However, one could randomize one of the two, lets say we randomise $v$. While we randomise $v$, $w$ becomes dependent on $v$ so that our guess does not fall outside of the triangle. Normally we would set $w$ to something similar to $w = 1 - v$, which will ensure that both $v$ and $w$ are inside the triangle. We can imagine this by setting $v = 1$, then $w = 1 - 1 = 0$ so we get the coordinate $(1, 0)$.

We can also see that the algorithm have some preset data (constants) that is set in the code. The first constant, $err$ is the initial max error that we set, we can usually set this high as the convergence rate of Newton’s method is fast. However, since we are working with a very low error this is set to 1. The second constant, $\epsilon$, is the error tolerance, more precisely it is the max on the interval for accepted error. For an exact solution $\epsilon$ would have
been 0, but Newton’s method is an approximation method we would assume that the method could result in some error. So the goal here is to keep the error tolerable low, this means however, that in some cases the intersection point could be a little bit off from the line segment. The last constant that is needed is \( t_{\text{max}} \), this tells the algorithm how many iterations should be tested before giving up. Usually we have around 3 – 4 iterations for each test, but if Newton’s method does not find any intersections/roots it can continue in infinite time if we did not have the max iterations set.

Further we let \( \frac{\partial b}{\partial v} \) and \( \frac{\partial b}{\partial w} \) be the derivatives of the basis functions, with respect to \( v \) and \( w \) respectively. We represent this in code in the same manner as the basis basis functions, meaning that we have two \( 12 \times 1 \) matrices which contains derivatives of the basis functions for \( v \) and \( w \).

The algorithm returns a \( 1 \times 3 \) matrix containing the coordinates of an intersection point. \( p^T \) is the Bezier patch that was provided as input transposed. \( b \) is the basis functions, that we have used. We base the result on the Bezier patch and basis functions, using \( v \) and \( w \) that Newton’s method have found.

An implementation of the algorithm \cite{Newton} is provided in the appendix Code: \cite{Code}. Since we mostly work with relatively small matrices we chose to use the method \texttt{colPivHouseholderQr()} from the Eigen library for solving linear systems in the implementation. \texttt{colPivHouseholderQr()} provides a great compromise between speed and accuracy for such small matrices like the ones that we work with. A complete comparison table of the decomposition methods is provided on the Eigen website\cite{Comparison}.

The basis functions that we use takes two parameters, \( v \) and \( w \) that represents coordinates in the Cartesian coordinate system. As we explained earlier the normal usage is the coordinates \((u,v)\) but since we are representing the unit triangle \((v,w)\) is used. We also include a third variable \( u \) which is the node of the triangle that has been mapped to origo, defined as

\[
u = 1 - v - w. \tag{3.11}\]

This means that \( u, v \) and \( w \) must be on the interval \([0,1]\). Because the functions \( s(v,w) \) and \( f(v,w) \) have the two parameters \( v \) and \( w \), we have to calculate the derivatives with respect to \( v \) and \( w \) respectively, resulting in two \( 12 \times 1 \) matrices. The two matrices contains derivatives of the basis functions at each vertex in the Bezier patch.

We also have the positions of each vertex in the Bezier patch, forming a \( 12 \times 3 \) matrix that we transpose, so it becomes a \( 3 \times 12 \) matrix. Then we multiply the two matrices which will result in a \( 3 \times 1 \). Lastly, we extract

\hspace{1cm} \footnote{http://www.eigen.tuxfamily.org/dox/group__TutorialLinearAlgebra.html}
the orthogonal coordinates from this matrix. The orthogonal coordinates are determined by the direction by the edge that we are computing the intersection point for. So as an example, if the edge is between two points on the $x$-axis we would extract the values equivalent to the $y$-axis and $z$-axis values in position 1 and 2 in the matrix. When we say that two points is located on the $x$-axis, we mean that only the $x$-coordinates are different while $y$ and $z$ remains the same.

After completion of this process, the resulting matrices (now of size $2 \times 1$) is placed within a Jacobian matrix, denoted $J$. Jacobian matrices are matrices that contain partial derivatives and forms a square matrix. Since our matrices containing the partial derivatives is $2 \times 1$, we get a $2 \times 2$ Jacobian matrix.

We now have a representation of both the function $f$ and also the derivatives in the Jacobian matrix $J$. The following equation defines a linear system and is considered.

$$Ax = b$$ \hspace{1cm} (3.12)

So in our implementation we use the equation above we can put in $J$ and $f$ so that we get.

$$Jx = f(v, w)$$ \hspace{1cm} (3.13)

The $x$ should, as we see be a solution in the form of a $2 \times 1$ matrix. We then use the C++ library Eigen \cite{2} to solve this linear system. Eigen is a rather extensive library for numerical computing, and thus we have quite a few options when it comes to solving linear systems. These have some advantages and disadvantages, which we have taken into account. Since we are testing the precision of our method, we want the accuracy of the solver to be as good as possible. Since we want to have the best possible precision of our results, we have to use a solver that is a little bit slower than a high-speed solver, which tends to have lower accuracy according to the Eigen documentation.

Algorithm \cite{2} shows how we traverse elements of the grid (FEM mesh) to compute intersection points using the algorithm based on Newton’s method. We notice that we differentiate on whether a patch is regular or irregular. Before the algorithm has begun, we check each element for whether and which triangles they intersect, triangles that do intersect an edge, is added to a list in the edge that contains pointers to all triangles that intersect with that specific edge. The first loops run through each element in the FEM mesh, for each of these elements we traverse each of its edges. Then, in the last inner loop we traverse each intersecting triangle of the edge. For each
Data: boundary elements of grid

for $e \leftarrow E$ to $n$ do
  $R \leftarrow \text{GetEdges}(e)$;
  for $r \leftarrow R$ do
    $T \leftarrow \text{GetIntersectingTriangles}(e)$;
    for $t \leftarrow T$ do
      $n \leftarrow \text{Valance}(t)$;
      if $n = \text{REGULAR}$ then
        $\vec{v} \leftarrow \text{ComputeNewton}(e, t)$
      else
        $t_1, t_2, t_3 \leftarrow \text{EvaluateIregularPatch}(t)$;
        foreach $t$ in $\{t_1, t_2, t_3\}$ do
          $\vec{v} = \text{ComputeNewton}(e, t)$
        end
      end
    end
  end
end

Algorithm 2: Workflow algorithm

of the triangles we check whether it is regular or irregular. If the triangle is regular we can start algorithm 1. However, if the triangle is irregular we need to split it so that we end up with regular triangles that algorithm 1 can handle.
Chapter 4

Results

Here we will present the results of tests that we conducted for the Loop’s subdivision surface/line segment algorithm that we showed earlier. We divide this chapter into three parts. In the first part, we will have a look at how the overall surface looks after applying the algorithm. Then, in the second part, we will have a closer look at how well the algorithm/Newton’s method does by comparing numbers and look at how large the error becomes. Lastly, we will look at potential problems discovered for the algorithm, these problems that need further investigation and maybe also fixes before one can say that the algorithm works in every case.

4.1 Surface Results

First, we shall look at the overall surface result after using the algorithm. We show in figure 4.2 a comparison of intersection points, one that shows the intersections using a traditional triangle/ray collision detection algorithm 4.2a and 4.2b, known as Möller–Trumbore intersection algorithm. Then we have figure 4.2c that shows the intersections after applying the algorithm presented. The two images can look very similar for the blind eye, which is good as the results should not be that off. However, the intersections should have shrunken a little bit. The reason to why we can conclude with that the intersections moves inward towards the centre of the sphere is the fact that the control surface is a convex hull for the limit surface. Since we the algorithm will compute intersections on the limit surface of the control surface.

In figure 4.2a and 4.2b we can see the resulting intersections using the Möller-Trumbore algorithm for ray/triangle testing. Which seems to be very accurate. Then in figure 4.2c we see that at first glance the intersection
points seems to be accurate, the intersection points seems to be at located at the surface.

4.2 Newton’s Method Results

Here we will look at the performance of the Loop’s subdivision surface/line segment algorithm, more specifically the part concerning Newton’s method in more details.

4.2.1 Radius Test Results

In figure 4.3 we have tested on the radius of the sphere vs. the distance from the center of the sphere to each of the intersection points. The green line indicates where the radius is, and the blue indicates the distance from the center of the sphere to the intersection points. For an optimal solution the blue and green line should be equal. We say that the difference is an error at intersection point. We see that in figure 4.3a the error is quite large for some of the intersection points, this figure contains irregular vertices in addition
to regular vertices. However, when we look at how large the error is, we can see that it is not very large and therefore may not affect the visualization for the human eye.

When we look at figure 4.3b we can easily see that some of the worst errors have disappeared, the reason for this is that intersection points for line segments going through irregular triangles have been filtered out/removed from the test. We can thus conclude that the method is somewhat unstable for irregular triangles.

A last notice we make us is how the error graph moves, we see that it begins where the error is higher than the radius and moves to the end where it gradually becomes less than the radius. By observation this means that some of the intersection points is located outside the sphere and on the other side we have some that is located inside the sphere.

4.2.2 Time consumption comparison

Here we will show a time consumption comparison between the more used triangle/ray intersection algorithm and the new surface/line Newtons’s method intersection algorithm. All the time comparisons was ran at the same system a Macbook Pro Retina late 2013. The implementation used for the Möller-Trumbore algorithm can be found in the appendix Code: Möller-Trumbore triangle/ray intersection algorithm implementation.

<table>
<thead>
<tr>
<th>Number of elements</th>
<th>Möller-Trumbore</th>
<th>Newton’s method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$5^3$</td>
<td>0.129ms</td>
<td>0.118ms</td>
</tr>
<tr>
<td>$10^3$</td>
<td>0.815ms</td>
<td>0.956ms</td>
</tr>
<tr>
<td>$25^3$</td>
<td>6.524ms</td>
<td>5.807ms</td>
</tr>
</tbody>
</table>

Table 4.1: Runtime time comparison Möller-Trumbore vs Newton’s method algorithm

As we see in table 4.1 the Möller-Trumbore algorithm is fast taken into account the number of elements used in the FEM mesh. While Newton’s method is a bit faster. This comes to light when the number of elements has a large increase, except at $10^3$ elements where the Möller-Trumbore algorithm is faster. To conclude, we see that the Newton’s method algorithm is a little bit faster or more or less have the same time consumption.

4.2.3 Convergence

Here we present the statistics and example of convergence rate for $f(v, w)$. We have used a tolerance of $\epsilon = 1e - 6$ and double precision for testing.
In table 4.2 we see that $f(v, w)$ converge towards 0, and reaches the tolerance eventually. In the test run done for the table it took Newton’s method 3 iterations before it reached the tolerance that was set.

If we were to test the convergence rate with a surface with no intersections, we would see that $f(v, w)$ does not converge and the iterations would continue until the maximum iterations is reached.
Figure 4.2: Examples of intersection tests done on Loop’s subdivision surfaces.
(a) Irregular vertices are included

(b) Irregular vertices have been filtered out. Only plot of regular vertices.

Figure 4.3: Plot of the radius vs distance from center of sphere to the intersection points.
Chapter 5

Conclusion and Future Work

By using the intersection algorithm between Loop Subdivision Surface and a line segment that we presented in this thesis we can say that it works very well for ordinary triangles, and where the line segment tested moves along one axis. As we saw, the algorithm requires some work on forehand before it can be used, such as the triangulation of the mesh needs to be compliant so that Bezier patches can be easily extracted when analyzed.

From the radius test 4.3 we saw that 4.3b had fewer significant spikes of error, which implies that the algorithm works better for regular triangles than irregular triangles.

5.1 Future Work

Here we present a few examples of future work that could be more interesting to investigate. Some of the suggestions listed here are concerned about integrating the implementation that we have presented in this thesis into existing software that could benefit from it. These suggestions are marked integration.

5.1.1 Speedup using parallelism

The Loop’s subdivision surface/line segment intersection algorithm should be able to run faster using various techniques of parallelism. Such techniques includes vectorization and SIMD operations for making iterations faster, such as the one we have in the Newton’s method part of the algorithm. Another technique is to multi-thread the algorithm in a suitable manner.
5.1.2 Extension to Quadrilateral Surfaces

Because of the popularity of triangular meshes nowadays we have chosen to focus on triangular meshes in this thesis. However, a possible extension to the implementation could be to adopt the methods used in this thesis to quadrilateral meshes.

The possibility of extending the methods presented in this thesis to quadrilateral meshes exists, thanks to Jos Stam paper [13]. This paper investigates the Catmull-Clark subdivision scheme as an example on evaluating the limit surface for quadrilateral meshes. As such, the Catmull-Clark subdivision scheme would be the best and also may be the fastest way to extend this method to quadrilateral meshes.

5.1.3 Improvements of the Subdivision Surface/Line Segment Algorithm

There are some some cases that has not been analyzed good enough in this thesis. These cases should be investigated in further details. Such case is when we have irregular triangles, which had an larger error.

Another thing that can be improved and will improve the performance of the algorithm is error fixing. By error fixing we mean that if an intersection point happens to fall a little bit outside the line segment, the algorithm should be able to correct the error.

5.1.4 Integration: Finite Element Method Solver

The purpose of our implementation is to make it in a fashion so that it can be integrated with existing FEM solver software.

One example of such software is the FEniCS Project[1].

5.1.5 Integration: Existing 3D Software

Even though the implementation presented in this thesis is impressive itself, it would be even more impressive to integrate it into a 3D modelling software. By doing this one could achieve faster model fixing.

Since a variety of 3D modelling software already exist on the market. A suggestion for future work will be that one could try to implement the application into one of the 3D modelling software. 3D modelling tools usually have an API that would make such integration possible. However, many of

[1] https://fenicsproject.org
the 3D modelling software out on the market is very expensive. But there exists at least one comprehensive 3D modelling software called Blender that is classified as open-source and is therefore also free to use. So a suggestion would be to integrate the implementation into Blender. Blender has an API for such customised integrations where one creates extensions for Blender itself using Python as a programming language.

The implementation would need extensive modification to work in case of converting it to Python code. Luckily libraries that make Python call functions written in C exists. An example of such library is the Cython library.
Bibliography


List of Figures

2.2  Catmull-Clark subdivision masks ......................... 22
2.3  Example of Loop’s subdivision surface .................. 23
2.5  Loop subdivision scheme masks ......................... 24

3.1  Indexing of Bezier patches ............................. 29
3.2  Half-edge structure ..................................... 30
3.3  Example of two AABBs intersecting ...................... 33
3.4  The unit triangle in vw-coordinates ..................... 37

4.1  The blue sphere is the geometry description and the grid is our FEM mesh ..................... 44
4.2  Examples of intersection tests done on Loop’s subdivision surfaces ..................... 47
4.3  Plot of the radius vs distance from center of sphere to the intersection points ............. 48
Appendices

Basis functions

\[ b^T(v, w) = \frac{1}{12} (u^4 + 2u^3v, \\
  u^4 + 2u^3w, \\
  u^4 + 2u^3w + 6u^3v + 6u^2vw + 12u^2v^2 + 6uv^2w + 6uw^3 + 2v^3w + v^4, \\
  6u^4 + 24u^3w + 24u^2w^2 + 8uw^3 + w^4 + 24u^3v + 60u^2vw + 36uvw^2 + \\
  6vw^3 + 24uw^2 + 36vw^2 + 12v^2w^2 + 8uw^3 + 6v^3w + v^4, \\
  u^4 + 6u^3w + 12u^2w^2 + 6uw^3 + w^4 + 2u^3v + 6u^2vw + 6uvw^2 + 2vw^3, \\
  2uw^3 + v^4 + 6u^3w + 12u^2w^2 + 6uw^3 + w^4 + 8u^3v + 36uw^2 + \\
  36uvw^2 + 8uw^3 + 24u^2v^2 + 60uw^2w + 24v^2w^2 + 24uw^3 + 24v^3w + 6v^4, \\
  u^4 + 8u^3w + 24uw^3 + 6v^4 + 6u^3v + 36u^2vw + 60uvw^2 + \\
  24uvw^3 + 12u^2v^2 + 36uvw^2 + 24v^2w^2 + 6uv^3 + 8v^3w + v^4, \\
  2uw^3 + w^4, \\
  2v^3w + v^4, \\
  2uw^3 + w^4 + 6uvw^2 + 6uw^3 + 6u^2w^2 + 12v^2w^2 + 2uv^3 + 6v^3w + v^4, \\
  w^4 + 2vw^3) \]
Code: Newton’s method

```cpp
RealVector3 GridElementEdge::NewtonIntersectionPatch(RealMatrixX patch)
{
    real tolerance = 1e-6, error = 1.0;
    RealVector2 f, vw(0.333, 0.333);
    RealMatrix2 J; // Jacobian Matrix
    RealMatrixX b, bv, bw;

    for (size_t i = 0; i < 100 && error > tolerance; i++) {
        b = Triangle::GetLoopBasisFunctions(vw);
        bv = Triangle::GetLoopBasisFunctionsV(vw);
        bw = Triangle::GetLoopBasisFunctionsW(vw);

        J.col(0) = GetOrthogonalCoords(patch.transpose() * bv);
        J.col(1) = GetOrthogonalCoords(patch.transpose() * bw);

        f = GetOrthogonalCoords(patch.transpose() * b
            mSourceNode > position);
        vw = J.fullPivHouseholderQr().solve(f);
        error = f.norm();
    }

    return patch.transpose() * Triangle::GetLoopBasisFunctions(vw);
}
```
Code: Möller-Trumbore triangle/ray intersection algorithm implementation

```cpp
real GridElementEdge::operator&(Triangle* triangle) const
{
    real tol = 1e-8 * this->GetLength();

    RealVector3 nodes[3] = {
        triangle->GetNode(0)->GetPosition(),
        triangle->GetNode(1)->GetPosition(),
        triangle->GetNode(2)->GetPosition()
    };

    RealVector3 edges[2] = {
        nodes[1] - nodes[0],
        nodes[2] - nodes[0]
    };

    RealVector3 pVec = mDirection.cross(edges[1]);
    real det = edges[0].dot(pVec);

    if (det < tol && det > tol) return 0.0;

    real invDet = 1.0 / det;
    RealVector3 tVec = RealVector3(mSourceNode->position - nodes[0]);
    real u = tVec.dot(pVec) * invDet;

    if (u < 0.0 || u > 1.0) return 0.0;

    RealVector3 qVec = tVec.cross(edges[0]);
    real v = mDirection.dot(qVec) * invDet;

    if (v < 0.0 || u + v > 1.0) return 0.0;

    return edges[1].dot(qVec) * invDet;
}
```

Code Repository

The full code of this thesis are available on Bitbucket. [https://bitbucket.org/kjetil92/subdivision-surfaces-finite-element/src/Subdivision/SubdivisionFEMProjectIntersections/](https://bitbucket.org/kjetil92/subdivision-surfaces-finite-element/src/Subdivision/SubdivisionFEMProjectIntersections/)