FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Integration of CAD models in the game engine Unreal Engine

**Bruno Miguel Vicente dos Santos**


Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Prof. António Coelho

Co-supervisor: Nelson Rodrigues

July 19, 2019

# Integration of CAD models in the game engine Unreal Engine

**Bruno Miguel Vicente dos Santos**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Doctor André Monteiro de Oliveira Restivo
External Examiner: Doctor João Paulo Fonseca da Costa Moura
Supervisor: Doctor António Fernando Vasconcelos Cunha Castro Coelho
July 19, 2019

# Abstract

Computer-aided design (CAD) software and 3D modeling software are similar, but they have different functionalities and applications. CAD software is a fundamental tool to create object models, design parts, and create 2D schematics from 3D designed objects, that can later be used in manufacturing. Comparatively, 3D modeling software is more used in entertainment, to create meshes for animation and games.

Despite CAD models not being intended to be used in game engines, there are times when being able to import them directly is advantageous, specifically when wanting to represent real objects in virtual and augmented reality environments. This creates the necessity of creating interoperability between CAD software and game engines, allowing to make use of models of real objects to create more immersive and detailed experiences.

The main concerns in importing models from CAD software into game engines are performance and quality of the resulting models. This is a trade-off that is hard to balance because rendering computer graphics is costly processing-wise, meaning too detailed models will result in poor performance, while at the same time too little detail will result in a worse user experience.

These limitations will be explored in this dissertation. Its goal is to create a tool that allows for the automatic creation of meshes in a game engine, based on the data contained in CAD files.

This dissertation will explore different methods for the creation of meshes and reduction of the number of polygons used to represent them, intending on giving the user control over how the model looks in the game engine, in a simple way. Besides this, it's also intended to optimize and simplify the mapping of textures to the created meshes.

In the end, an application was created that generates accurate 3D models based on CAD surfaces, while giving the user more control over the final result than other current solutions.

# Resumo

Software de desenho assistido por computador (CAD) e software de modelação 3D são semelhantes, mas têm diferentes funcionalidades e aplicações. Software CAD é uma ferramenta fundamental para criar modelos de objetos, projetar peças e criar esquemas 2D a partir de objetos projetados em 3D, que podem ser usados posteriormente na indústria. Comparativamente, o software de modelação 3D é mais usado em entretenimento, para criar malhas poligonais para animação e jogos.

Apesar dos modelos CAD não se destinarem a ser usados em motores de jogos, há momentos em que ser capaz de importá-los diretamente é vantajoso, especificamente quando se deseja representar objetos reais em ambientes de realidade virtual e aumentada. Isto cria a necessidade de criar interoperabilidade entre o software CAD e os motores de jogo, permitindo fazer uso de modelos de objetos reais para criar experiências mais imersivas e detalhadas.

As principais preocupações na importação de modelos de software CAD para motores de jogos são o desempenho e a qualidade dos modelos resultantes. Este é um balanceamento que é difícil equilibrar pois a renderização de gráficos de computador é dispendiosa em termos de processamento, o que significa que modelos muito detalhados resultarão em pior desempenho e, ao mesmo tempo, detalhe reduzido resultará numa pior experiência do utilizador.

Essas limitações serão exploradas nesta dissertação. O seu objetivo é criar uma ferramenta que permite a criação automática de malhas num motor de jogo, baseado nos dados contidos em ficheiros CAD.

Esta dissertação irá explorar diferentes métodos para a criação de malhas e redução do número de polígonos utilizados para representá-las, procurando dar controlo ao utilizador sobre as características do modelo no motor do jogo, numa forma simples de utilizar. Além disso, também pretende otimizar e simplificar o mapeamento de texturas para as malhas criadas.

No final, foi criada uma aplicação que gera modelos 3D de forma precisa, com base em superfícies CAD, dando, ao mesmo tempo, mais controlo ao utilizador sobre o resultado final, comparativamente as outras soluções existentes.

# Acknowledgements

I would like to thank my supervisor, Professor António Coelho, as well as Abyssal, specifically Nelson Rodrigues, Pedro Costa and Pedro Maia, for their continuous support throughout this dissertation.

I would also like to thank my parents and my friends for their love and support.

Bruno Santos

# Contents

# List of Figures

# List of Tables

LIST OF TABLES

# Abbreviations

| | |
|---|---|
| AI | Artificial Intelligence |
| AR | Augmented Reality |
| ASCII | American Standard Code for Information Interchange |
| BFF | Boundary First Flattening |
| CAD | Computer-Aided Design |
| COLLADA | Collaborative Design Activity |
| CSS | Cascading Style Sheets |
| DOM | Document Object Model |
| FBX | Filmbox |
| IGES | Initial Graphics Exchange Specification |
| IPC | Inter-Process Communication |
| ISO | International Organization for Standardization |
| HTML | Hypertext Markup Language |
| MIP | Multum in Parvo (Much in little) |
| NURBS | Non-uniform rational basis spline |
| RMI | Remote Method Invocation |
| ROV | Remotely Operated Vehicle |
| RPC | Remote Procedure Call |
| SDK | Software Development Kit |
| STEP | Standard for the Exchange of Product Data |
| STL | Standard Triangle/Tessellation Language |
| SWIG | Simplified Wrapper and Interface Generator |
| VR | Virtual Reality |
| XML | Extensible Markup Language |

# Chapter 1

# Introduction

## 1.1 Motivation and context

One of the main processes when creating a 3D application with a game engine is asset creation. The objects displayed on the screen by a game are usually built by designers in modelling software from scratch. This type of applications give the user the ability to sculpt objects to match their needs and artistic vision.

On the other hand, CAD models describe objects that are supposed to be created and manufactured in real life instead of just being digital. Because of this, these models have to be mathematical in nature, assuring the rigor necessary when handling assembly and all of its related physical properties.

Abyssal[1] is a company that provides 3D visualization, simulation and digitization capabilities for sub-sea operations. Among other solutions, the company develops augmented reality software, using Unreal Engine, for the remote control of sub-aquatic maintenance drones, operated in conditions of reduced visibility.

Abyssal proposed this project with the objective of bridging the gap between CAD and the 3D representation of models in game engines. Their interest in this subject comes from the fact that, when re-creating the underwater environments where operations take place, they have to model all of the objects present in the area. These objects are usually industrial parts that had to be manufactured and, thus, have CAD models. By creating an application that can import these CAD models directly into a game engine, the process of modelling can be eliminated or reduced from the workflow, thereby increasing development speed.

This dissertation is focused in Computer Graphics, and more specifically it relates to computational geometry and modeling.

[1] https://abyssal.eu/

## 1.2 Objectives and problem statement

The main objective of this dissertation is the creation of an application that can import CAD models into Unreal Engine.

The challenge of creating 3D meshes from CAD models comes from the fact that meshes are discrete in nature, while CAD models are continuous. To represent an object, CAD utilizes mathematical functions that define surfaces and edges. Contrarily meshes are defined by a discretization of its underlying surface into a finite number of vertices and polygons, an operation that is done to optimize performance when rendering graphics in real-time applications.

There already exist some solutions that provide controlled mesh creation from CAD models, but the results obtained from them are not up to Abyssal's standards. They also mentioned the lack of user interaction in these applications and the inability to affect the creation of the mesh in a user friendly way.

As such, an important aim of this project is the creation of low polygon meshes that can still preserve the features of the underlying surfaces, minimizing the visual artifacts that come from the triangulation of these surfaces. Further, this application has the intention to provide the user a way to control how the mesh is created and interactively modify the outcome of the process.

A secondary objective of this thesis is the development of a method that can create texture coordinates automatically for the generated meshes. Texturing is a simple and effective process, which can increase the realism of meshes at a low cost by applying an image, or texture, to it. Since the created meshes are 3D objects, while images belong to a two-dimensional domain, a parameterization process is required to assign 2D coordinates to 3D vertices. This dissertation will explore how this mapping is done, in order to obtain UV coordinates that minimize both discontinuities and distortion of the used texture.

## 1.3 Methodology

The approach to the development of this project was to first gather all of the needs Abyssal had regarding the integration of CAD models into Unreal Engine, and what features would be necessary in a proposed solution. Then, it was fundamental to research and understand concepts related to the area, and more specifically to the representation of CAD models and 3D meshes. At the same time, it was necessary to survey related work and available libraries related to mesh generation and surface parameterization. These initial processes allowed for the architecture of the application to be defined, already taking into account the integration of external modules to speed up development.

The first step of the development process was to created an application that could import a STEP file, provide the user a way to visualize it, and export it to an FBX file.

From there, the objective was to focus on user interaction, giving the user control over how the different parts of a model were tessellated, and providing functions such as selection, and part, polygon and normals inspection.

The final step of the development process was the creation of a method that allowed automatic UV generation, as well as giving the user the capability to affect how the mapping was done.

In the end, the results of the application were compared to the results obtained using the tool previously used at Abyssal, Datasmith. The designers of the company also evaluated the final solution, taking into account the fulfilment of the set requirements, and the performance and usability of the application.

## 1.4 Structure of dissertation

Besides the introduction, this dissertation contains five more chapters.

Chapter 2 describes and exposes relevant background knowledge required, as well as presents related work.

Chapter 3 lists the requirements of the project, describes the main workflow of the application and explains the more methodical side of the project.

Chapter 4 presents the architecture of the application and goes into detail about how each feature was implemented.

Chapter 5 explores the results obtained with the developed application, compares them to the results of other existing tools and presents an evaluation of the application done by Abyssal's designers.

Finally, Chapter 6 presents conclusions and future work.

# Chapter 2

# Literature Review

To implement the project proposed in this dissertation multiple fundamental concepts had to be explored and understood. As such, the first part of this chapter (Section 2.2) will provide an overview of these concepts. The second part (Section 2.3) will focus on implemented solutions related to the project proposed, with emphasis on the two main topics of this dissertation: discretization of CAD models and automatic UV mapping.

## 2.1 Fundamentals

### 2.1.1 Computer assisted design (CAD)

The use of computers revolutionized many areas, including design and manufacturing. Computer-aided design and drafting provided a better way to create, modify, analyze and collaborate on designs, leading to an increase of productivity and improved quality compared to the previous manual processes. CAD files convey information such as shape, dimensions, tolerances, materials and processes of an object in a accurate way. This makes it an indispensable tool for industries such as manufacturing and construction [MM11].

There are multiple types of CAD formats, with different types being used for different purposes. Two-dimensional models are often standard in manufacturing and construction. These are quicker to design and present limited visualization and simulation features, providing multiple 2D views of an object. Three-dimensional models can have different levels of detail. Wireframe models are the most basic and only contain information about edges and vertices. Surface models provide information about edges, vertices and surfaces (outer boundaries of objects), as well as being able to display color, shading and textures. Solid models are the most complex CAD models, being also the most common format used currently. A solid model is similar to a surface model as it has all of the same features, but instead of using zero thickness elements, solid models use volumes to represent a product. These volumes can also have mass associated to them, which

allows analyzing and simulating an object's physical properties. This results in accurate digital representations of products [MM11].

CAD software provides the user with multiple ways to create and modify solids. These can be Boolean operations (union, difference and intersection) on primitive solids, extrusion, chamfering or blending of surfaces, or changing controls points of NURBS curves. These operations are mathematical in nature, and CAD uses parametric mathematical functions internally to represent the surfaces of an object. This means that CAD models use a continuous domain [MM11].

#### 2.1.1.1 CAD file formats

As stated by [BO91], interoperability between CAD systems was a problem. The difference in representation of the entities between systems led to the need to define standards for CAD files.

ISO 10303, also known as STEP, is a standardization effort to define rules for the representation and exchange of the product manufacturing information of 3D objects created using CAD software [ISO95]. STEP files contain definitions of products, surfaces and materials, using an ASCII structure, which facilitates use and exchange.

Each line of a STEP consists of an entity with a unique numerical ID and a definition. Entities can reference other entities, creating a tree-like structure. This means a product will reference multiple solids, solids will reference their surfaces, surfaces their lines, and lines their control points, for example. A product is obtained by assembling parts together, with the hierarchy and relationship between parts being easily represented using STEP [DRV98]. An extract of a STEP file can be found in A.

There also exist other file formats to represent CAD files. For open formats, IGES is an older format (1980s as opposed to STEP's 1990s) and is mostly used for surface models. Since it has become outdated and unsupported, importing and exporting using IGES can result in missing geometry. There also exist multiple proprietary formats, usually associated with specific CAD applications, like Solidworks or AutoCAD.

### 2.1.2 Game engine basics

Game engines are software that provide creators an environment and a set of features that allow them to create video games. Usually, these features are elements like 3D graphics rendering, audio, physics, AI and networking [Gre14].

Different game engines may provide different features or be built in different programming languages. Despite this, the underlying architecture of games end up being similar. Games are real-time, dynamic computer simulations and, because of this, the concept of time is incredibly important. A game loop, physics loop, or logic loop, is the main process behind the execution of a game. This periodically updates the game state, like the position of assets and how they interact. Concurrently, there is usually a render loop, which is in charge of how and what should be displayed on the screen, and how a scene is translated to pixels on the screen [Gre14].

A game's scene is usually defined in 3D space and is composed by a camera, whose position defines the point of view of the scene and visibility of its objects; one or multiple light sources, whose light rays interact with and reflect off the objects in the environment; and multiple objects, each with a specific shape, position and material, which defines the color of an object and how light should interact with it [Gre14].

While the game's logic loop updates the scene's attributes, such as positions, periodically, a game's render loop is in charge of calculating what objects are visible at a given time, the interaction between light and objects, and the resulting color to be displayed on the screen. These calculations are done periodically, with each new tick of a render loop originating a new frame. These updates usually occur around 60 times per second, as that is the most normal refresh rate of monitors, but this is highly dependent on how complex are the calculations done at each frame. The more intricate the geometry of the objects to be displayed, the more difficult it is to make these calculations. As such, the rendering of computer graphics become a trade-off between visual fidelity and performance, which has to be managed to obtain the best results [Gre14].

### 2.1.3 Computer Graphics in CAD applications and game engines

Like referenced before, CAD applications store scene objects by the mathematical functions that define them. This allows a high degree of accuracy and good translation into real-life manufacturing. This high fidelity also means it's extremely complicated to analyze and display these shapes directly. Because of this, CAD surfaces are translated from their original continuous domain, into a discrete sampled domain.

Much like in CAD, game engine scenes are displayed on the screen by sampling them into a 2D pixel grid. Unlike CAD models, the internal representation of the objects in the scene is done using polygonal meshes, and more specifically triangular meshes. The composition of polygons in a mesh serve as a linear approximation of their underlying surface, just as a series of connected line segments serve as a approximation of a curve [Gre14]. Triangular meshes specifically are widely used because triangles are automatically convex, planar and easy to interpolate, making geometric transformations, and color and lightning calculations easier [FvDFH90]. A mesh representation is composed by vertices, and their positions; faces, and what vertices compose them; and normal vectors that give information about the orientation of a face at each of its vertices.

### 2.1.4 Tessellation

The process of dividing a surface into a collection of discrete polygons is called tessellation [Gre14].

Quad meshes are meshes in which the faces uses are quadrilateral. Quad meshes are preferred for many applications because they capture shape features better, as quads can be aligned with the principal and secondary curvature of a shape, while triangles require an arbitrary third edge direction. The edge flow introduced by quad meshes mean that textures can be mapped with

ease, with potential seams following the orientation of the surface, resulting in better looking discontinuities [BLP+13].

Triangulation refers to the tessellation of surfaces into triangles [Gre14]. Triangular meshes are typically characterized by being unstructured and having greater freedom in the placement of nodes, which allows to significantly reduce the number of vertices used to represent a surface, while maintaining the same accuracy regarding the distance to it [She99].

There are three different approaches to generating triangular meshes identified by [Bak05]: the moving/advancing front technique, Delaunay based methods and the Octree approach.

#### 2.1.4.1 Moving front method

The moving front method (2.1) starts from a set boundary. The boundaries of the surface are first sampled, and interior nodes are generated based on the spacing of the boundary nodes. The boundary between the meshed and unmeshed region advances by choosing a new interior node and adding it to the mesh. This node is chosen based on constraints imposed by other parts of the advancing front, as so to avoid overlaps, and by the quality of the resulting triangle. The quality of a generated triangle is given by the difference between its angles, with it being better the closer it is to an equilateral triangle. In the end, a smoothing process is applied by shifting each interior generated node to the center of the surrounding polygon [Lo85].

This method can have difficulties in its closing stages when the front collapses on itself, as the unmeshed part can have an extremely complicated shape that results in less than ideal triangulation [Bak05].

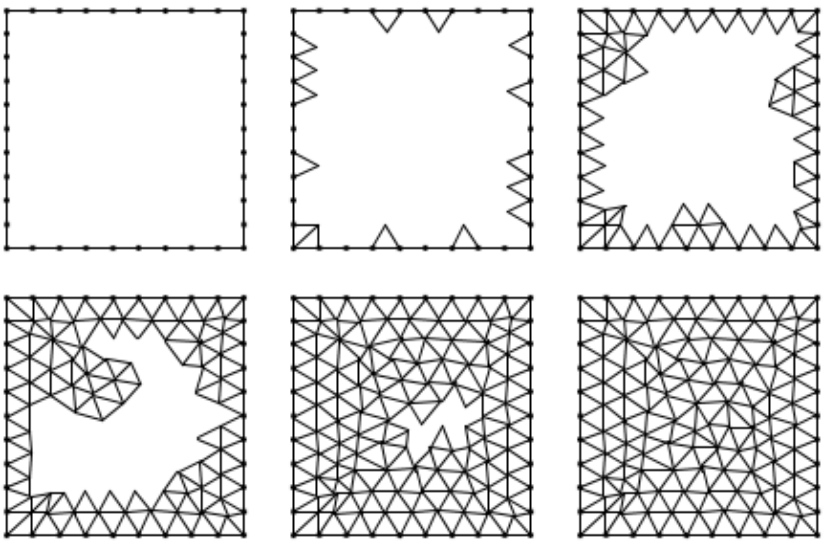

Figure 2.1: Multiple iterations of the moving front method

#### 2.1.4.2 Delauney methods

Delaunay triangulation (2.2) aims at maximizing the minimum angle of all of the angles of the triangles in the mesh, thus avoiding the creation of triangles with very acute angles. It accomplishes this by computing the circumcircle of every triangle of the mesh. After this, new points are introduced, making sure that no point is inside the circumcircle of any triangle. The triangulation can be locally reconstructed to fit this constraint, generally done by "flipping" the common edge between two invalid triangles [Bak05].

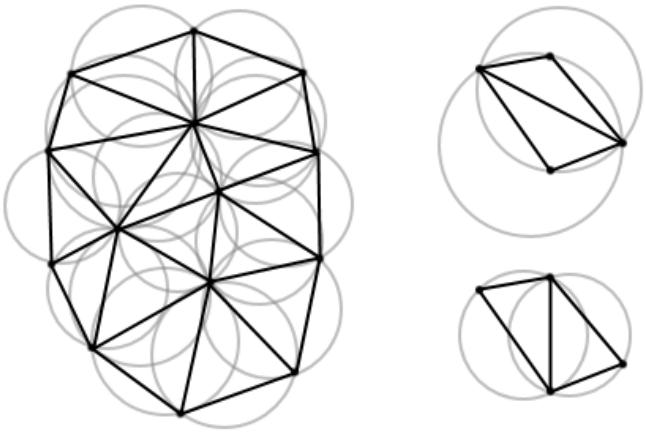

Figure 2.2: Delauney triangulation with circumcircles and example of an edge flip

A Delaunay triangulation can be further refined as stated by [She02]. Ruppert's algorithm is an example of a Delaunay refinement, where new vertices are added and triangles redone in such a way that the final result has no poor-quality triangles - triangles where the ration between radius of the circumcircle and their shortest edge is larger than a certain threshold.

The Delaunay method can have a problem ensuring surface integrity, as the introduction of points in an unchecked fashion can lead to the change of surface boundary edges [Bak05].

#### 2.1.4.3 Octree decomposition

Octree decomposition starts off with a grid of cubes. When they intersect with the boundary of the surface, these cubes can then be subdivided recursively into eight octants. These subdivisions can be done until the desired resolution is obtained. The intersection of each cube with the surface can then be replaced with triangles [Bak05]. For a two-dimensional domain, the equivalent of this method is called quadtree decomposition (2.3), since it uses squares and their division is done by four.

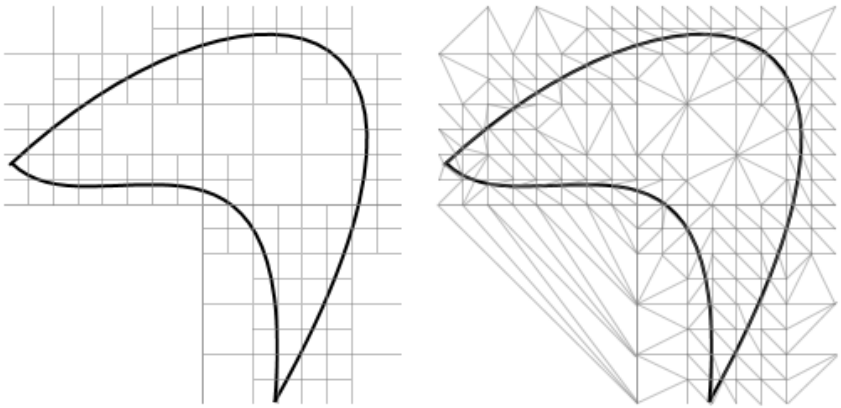

Figure 2.3: Division and triangulation of a 2D surface by the quadtree method

### 2.1.5 Mesh simplification

When generating a mesh, there exists a trade-off between the number of polygons used and the performance of the mesh when used in real-time environments like game engines. This is especially true when multiple objects are visible at the same time, meaning some sort of optimization has to be done in order to increase performance [LT97].

Frequently, there exist vertices and faces in a mesh that are visually redundant. When the underlying mathematical surface of a mesh is known, this issue can be resolved by re-meshing the surface with a higher error tolerance. But when this is not the case, mesh simplification algorithms have to be applied to try to obtain a good approximation. These algorithms work by removing vertices and reconstructing polygons into larger ones, reducing the geometry used to represent a mesh, while keeping perceptual difference between original and simplified at a minimum [III04].

#### 2.1.5.1 Vertex decimation

Vertex decimation works by selecting and deleting a single vertex, and re-tessellating the resulting hole.

For each vertex of the mesh, this algorithm computes the plane created by its adjacent vertices. The weight associated with the removal of a vertex is the distance of said vertex to this plane, with the vertex with the lowest weight being removed each iteration.

This guarantees that vertices in smooth regions are removed before vertices that define sharp features [SZL92].

#### 2.1.5.2 Edge collapse

Edge collapse contracts an edge into a single vertex. For every edge of a mesh, the vertex that an edge is collapsed into resides alongside that same edge. Its position can be coincident to either one of the vertices that constitute the edge, or be calculated based on an error function, which uses parameters such as its distance to the original mesh, or its preservation of sharp features of the

mesh. Once the position of the vertices associated with every collapse is computed, the chosen contraction is the one with the lowest error associated with it [Ros97].

#### 2.1.5.3 Vertex clustering

Vertex clustering starts off by grading each vertex of the mesh by its perceptual importance. This means that vertices in high density, smooth areas are weighted lower than vertices in low density, high curvature areas. The 3D space that contains the mesh is then divided into a grid, with the vertices inside each grid cell being clustered into a single vertex. The positions of the generated vertices is given by the weighted interpolation of the vertices of its cell [RB93].

This is an efficient global strategy for simplifying meshes, with the level of simplification being able to be controlled by modifying the resolution of the grid. Despite this, the results obtained aren't the best as it can drastically alter the topology of the mesh in an unpredictable manner [III04].

#### 2.1.5.4 Comparison between mesh simplification algorithms

For vertex decimation, the position of the vertices is fixed. In the example of an octagon, if vertex decimation were to be applied, the result would be an heptagon with one side longer than all of the others. Extrapolating this example to 3D space, it means that parts with regular curvature, like cylinders, would have an irregular tessellation on multiple iterations of the algorithm.

For edge collapse, even though new vertices can be created, the same problem is faced as in vertex decimation. Due to its local nature, the reduction of an edge results in irregular tessellation. In the previous example of an octagon, edge collapse would generate a heptagon with two sides longer than the rest.

Vertex clustering, being a global method, allows to tackle the uniformization problem. Despite this, it also introduces other problems like the heavy alteration of topology and the excessive reduction of geometry each iteration.

A visual comparison between the three methods can be seen in 2.4.

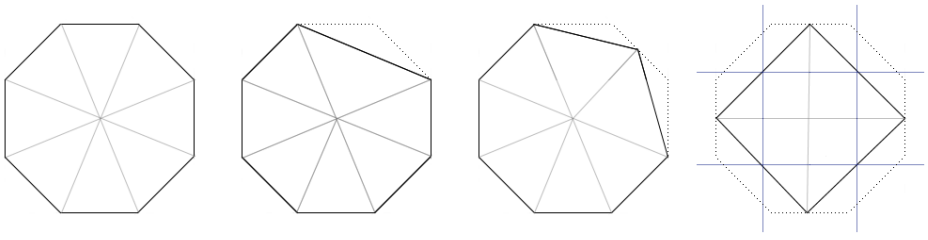

Figure 2.4: Comparison between mesh simplification algorithms on an octagon
From left to right: original, vertex decimation, edge collapse, vertex clustering

### 2.1.6 Topology basics

Topology is the study of properties that remain unchanged when distorting an object by a continuous deformation. A continuous deformation is any geometric deformation that stretches, shrinks, twists or bends a shape, without tearing it, breaking it apart or joining it to another shape [Tan14].

Topology differs from geometry because it ignores metrics such as distance between points or angles of edges, instead focusing on the relative position of points in a surface. If a surface is considered as a set of adjacent, connected points, transformations such as scaling or changing the shape of its borders preserve the underlying connectivity of these points despite changing the overall shape of the surface. Transformations such as tears change the local neighbourhood of the points alongside the seam and, thus, do not preserve topology [Fle01].

When a surface can be obtained from another through the application of topology preserving transformations these two surface are said to be topologically equivalent, or homeomorphic (2.5). For example, a square is homeomorphic to a circle by the stretching of its borders, or a mug is homeomorphic to a torus [Fle01].

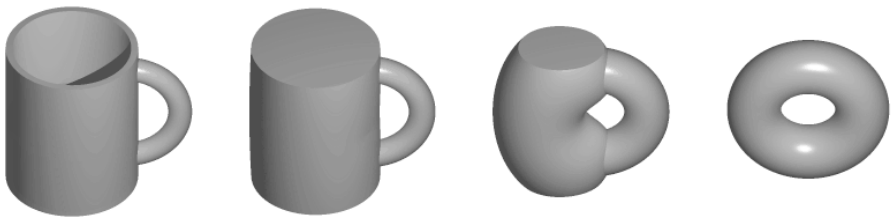

Figure 2.5: Homeomorphism between mug and torus (Source: Wikipedia)

The Euler characteristic is a topological invariant that describes a shape's topology, with shapes with the same Euler characteristic being homeomorphic to each other. The formula for the Euler characteristic is given by:

$$\chi = V - E + F \tag{2.1}$$

with V, E, and F referring to the number of vertices, edges and faces of the shape. A hole in the surface of a shape affects its Euler characteristic by -1, as it can be equated to a missing face.

### 2.1.7 Textures basics

Texturing is a powerful method to add visual detail to a model without introducing geometric complexity. This process is done by assigning UV coordinates to each vertex of a mesh, with these coordinates corresponding to a point of the plane containing a texture image [FvDFH90]. A texture atlas refers to the packing of multiple texture images into a single texture. This allows for all of the textures associated with a single object to be stored in a single file, reducing texture fetches, which can improve frame rate.

When a mesh is displayed on the screen, each pixel can correspond to part of a texel (unit texture cell), or to multiple texels. The former results in the magnification of the texture leading to blurriness. The latter may create an aliasing effect, as sampling of widely separated texels may result in the creation of visual artifacts [Cun06].

One solution for the previous problem is the use of filtering functions, that determine the color of a pixel based on, for example, the weighted average of the surrounding texels. Another solution is the use of MIP mapping, an approach that downsizes the texture image to multiple different resolutions. What resolution is mapped to what polygon depends on the ratio between pixel and texel. Besides these two methods, it is also common to insert padding between different elements in texture atlases. When multiple texels are contained in a single pixel and sampling is done, the existence of padding makes it so that the color of the texels of a different subdivision of the mesh don't disturb the current subdivision [Cun06]. This can be seen in 2.6.

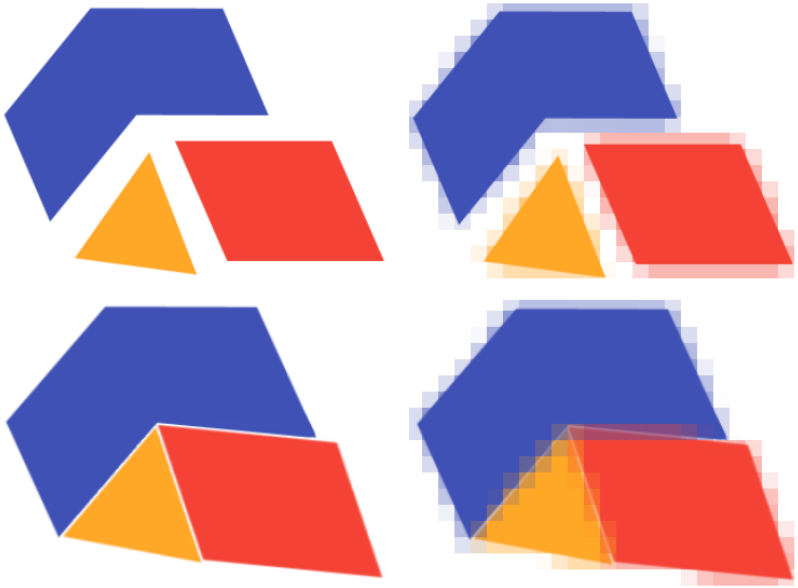

Figure 2.6: Difference between padded and unpadded texture atlases when sampling

### 2.1.8 Mesh parameterization

Meshes are usually defined in three-dimensional space, while texture images belong to a two-dimensional domain. Mesh parameterization is the name given to the process of calculating and assigning UV coordinates to vertices of a mesh.

A parameterization of a 3D surface to a 2D one will almost always introduce some sort of distortion, either by the angles between polygons, or by their areas, with parameterization algorithms trying to minimize these distortions in one way or another [SFH05].

In graph theory, a planar embedding of a graph is the representation of a graph on the plane where its edges intersect only at their endpoints. In 1963, Tutte proposed the embedding of a graph

onto the plane by mapping its outer face, or boundary, into a convex polygon, with the coordinates of interior vertices being calculated as the average of its neighbours' positions [Tut63].

Since the connectivity of a triangular mesh can be equated to a graph, Tutte's method can also be applied to meshes topologically equivalent to disks. Even though the mapping created is bijective (function f: X $\rightarrow$ Y is one-to-one and surjective), it doesn't preserve shape properties rendering it a poor method [SPR06].

Since Tutte, multiple efforts have been made to develop algorithms to parameterize triangular meshes in a way that optimizes preservation of areas, angles or distances. This process usually involves the minimization of an energy function, which balances angle and area distortion, and the resolution of a linear system that, among other things, takes into account the curvature of the mesh at each vertex, to compute the final UV coordinates [SFH05].

### 2.1.9 Texture atlas packing

As seen in section 2.1.7, packing the textures of the subdivisions of a mesh is a common practice to reduce load time. An important part of this process is how the texture images are arranged, because the more empty space is left in a texture atlas the lower its size will be.

Even though UV patches usually have irregular boundaries, reducing each of them to their bounding box makes their packing easier and turns this problem into a two-dimensional rectangle bin packing problem.

Two-dimensional bin packing is the problem of packing a finite set of rectangles into the minimum number of containing bins, with each bin having a limit of area, weight or other cost associated with it. In the context of texture packing, the object of the algorithm is to minimize the area of the container that holds all of the parts of the texture [LMV02].

#### 2.1.9.1 Shelf algorithms

These algorithms are the simplest algorithms for bin packing. Shelves are horizontal sub-divisions of the bin that are filled left to right, rotating rectangles in order to minimize first the height of each shelf, and then its width (2.7). How the packing is done specifically depends on the variation of the algorithm [Jyl10]:

- **Next Fit** tries to fit the rectangle into the current shelf, creating a new one if it doesn't fit width-wise.
- **First Fit**, unlike Next Fit, keeps all of the shelves open, with each new item being inserted into the first shelf where it fits.
- **Best Width Fit** selects the shelf where to insert a rectangle by minimizing the remaining width after insertion.
- **Best Height Fit** optimizes for lowest remaining vertical space.
- **Best Area Fit** optimizes for lowest remaining area.

In general Next Fit produces the worst results as it doesn't worry about optimizing rectangle placement, while the rest of the alternatives produce similar results, with the best depending on situation [Jyl10].

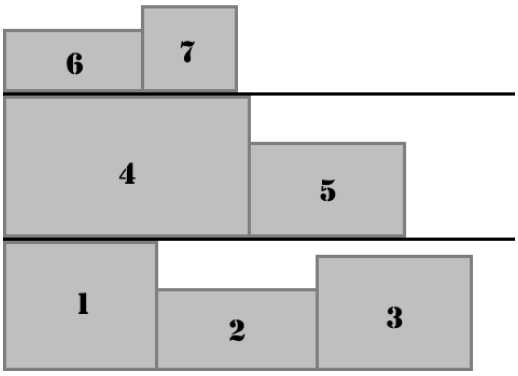


Figure 2.7: Packing produced by a Shelf algorithm, with subdivisions in black

#### 2.1.9.2 Guillotine algorithms

Instead of using horizontal divisions, guillotine algorithms (2.8) split the unused area into two, either horizontally or vertically, every time a new rectangle is inserted. One of the splits will therefore have either the width or the height of the inserted rectangle [Jyl10].

When inserting a new rectangle, the disjoint spaces created by previous insertions and subsequent divisions are analyzed, and the rectangle is inserted in the space that either minimizes remaining area, width or height [Jyl10].

The choice of axis of how to cut the remaining space after insertion can also be different, with some implementations depending on the width/height of the inserted rectangle, others on the remaining width or height of the remaining space, and others minimizing or maximizing the area of the spaces [Jyl10].

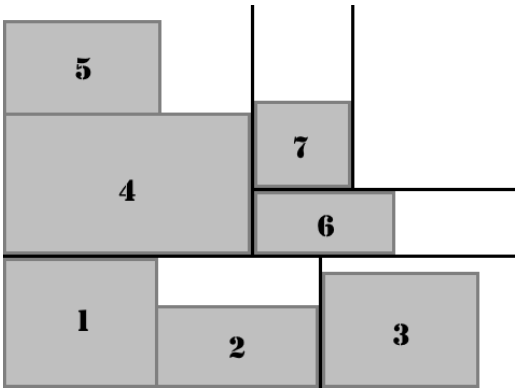


Figure 2.8: Packing produced by a Guillotine algorithm, with subdivisions in black

#### 2.1.9.3 Maximal Rectangles algorithms

The maximal rectangles algorithm (2.9) can be seen as an extension of the guillotine algorithms. Instead of choosing to divide the free space either horizontally or vertically, both divisions are done, with the available spaces becoming the maximum area part of each division. This results in two partly overlapping spaces [Jyl10].

The selection of spaces where to insert new rectangles depend on similar aspects to the previous algorithms (best area/side fits). On insertion not only is the space where the rectangle was inserted updated, all of the other free spaces have to be checked for intersections with the new rectangle and updated [Jyl10].

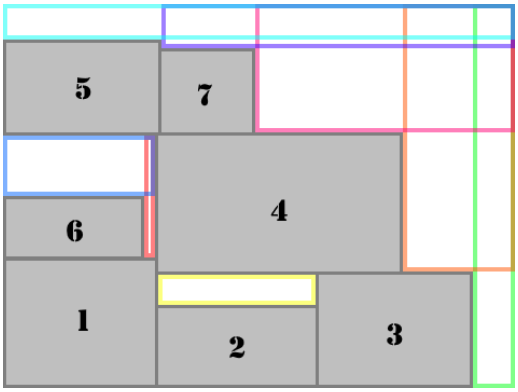


Figure 2.9: Packing produced by a Maximal Rectangles algorithm, with free spaces in color

### 2.1.10 3D mesh file formats

There exists a multitude of 3D modelling applications available and, associated with most of them a proprietary 3D file format. There are over 140 file formats to store 3D data, most of them proprietary. This severely hinders interoperability between applications, with imports, exports and conversions resulting in problems like missing or inverted faces, wrong scale of objects, or invalid surfaces [MB08].

Some of the more used 3D model file formats are:

- **OBJ** - open file format in ASCII that can only store 3D geometry: vertex positions, vertex normals, faces and UV coordinates. Faces and surfaces can be joined into groups for easier editing and animation of 3D models but they can't have a specific hierarchy. Materials require an additional file to be stored (.mtl file) [OBJ].
- **STL** - open format that can be both in ASCII or binary. It is one of the simplest formats, only encoding surface geometry [STL].
- **COLLADA (.dae)** - it's an open format that uses an XML schema. It can store scene information, object hierarchy, geometry, materials, animation, shaders and physics [Inc08].

- **FBX** - proprietary file format by AutoDesk. Supports scene information and hierarchy, geometry, materials animations and morphs. Widely used in the game industry due to its complete representation of scenes and the popularity of AutoDesk software. AutoDesk provides an SDK for the reading and writing of FBX files [FBX].

## 2.2 Related work

### 2.2.1 CAD to geometry conversion

While the study of surface meshing spans several decades, the integration of CAD models into game engines seems to be an under-explored area. Even though multiple CAD applications allow for the exporting of STL or FBX, and there exist some CAD converters associated with game engines, they often lack finer ways to control how tessellation is done. The emergence, in recent years, of solutions that provide interoperability between CAD models and game engines indicate a rising need for this process, as the uses of AR and VR systems within the industrial and construction sectors is more researched and developed.

#### 2.2.1.1 Datasmith

Datasmith[1] is a built-in plugin of Unreal Engine that was first introduced in 2017, and is currently in beta testing.

Datasmith reads many common CAD file formats, both open formats like STEP and IGES, but also proprietary formats like the ones used in CAD applications like SolidWorks and 3ds Max [Data].

The main drawback of Datasmith is how the user controls tessellation of the object. Datasmith allows control over three parameters [Datb]:

- Chord tolerance - the maximum difference from the tessellated surface to the original surface
- Max edge length - the maximum distance between two adjacent vertices in the resulting mesh
- Normal tolerance - the maximum angle between two adjacent triangles in the mesh

These parameters are set on import, which means that if the user isn't satisfied with the obtained tessellation, either because of performance or visual issues, the object needs to be imported again. This can become a slow, tiresome, trial-and-error process, especially with bigger sized files.

Another drawback of Datasmith is that the meshes aren't imported with a UV mapping, with the user having to export the generated mesh into a file, manually map its UV coordinates in an external application and re-import it into Unreal.

[1] https://docs.unrealengine.com/en-us/Studio/Datasmith

#### 2.2.1.2 Theia Optim

Optim[2] is an Unreal Engine plugin by Theia, developed on top of Datasmith and, like Datasmith, is also in beta [The].

Optim provides visualization tools for an easier analysis of a generated mesh, displaying triangle count, distribution and scale, and material and light lists, among others [Opt]. It also allows the creation of rules to optimize imported meshes. These include the combination of actors, or the defeaturing of the surfaces to remove irregularities like small holes, protrusions or unseen geometry, thus increasing performance without sacrificing visual quality.

#### 2.2.1.3 PiXYZ

PiXYZ is a company focusing on CAD data, mesh generation and optimization [PiXa]. They have two main products related to mesh generation from CAD files: the PiXYZ Plugin, and the PiXYZ Studio.

The PiXYZ Plugin is available for both Unreal Engine and Unity and works much like Unreal's Datasmith. The user selects the CAD file and what tessellation quality they want and a 3D mesh is generated and imported into the game engine [PiXb].

The PiXYZ Studio is a standalone application that can import CAD files and export the desired mesh file format. It features tessellation by parts, hole removal, decimation of vertices and provides repair functions such as removal of duplicated faces or normal orientation unification [PiXc].

Tata Elxsi compared PiXYZ Studio with Unreal Datasmith by benchmarking the two on the same tasks [PiX18]. The benchmark compared the workflow of the steps like importing, mesh repairing, application of materials and decimation on metrics such as quality of the results and execution time. The workflow for Datasmith also included the use of 3ds Max to handle the processes that could not be accomplished using only Datasmith. The results of experiments can be seen in 2.1, with more details being available in [PiX18]. PiXYZ obtained better results across the board, but it has to be noted that both this benchmark report and PiXYZ are sponsored by Unity Technologies.

Table 2.1: Comparison between PiXYZ and Datasmith (abridged) [PiX18]

| Functionality | PiXYZ | Datasmith |
|---|---|---|
| Import STEP data | 227 seconds | 390 seconds |
| Applying materials | 5 hours | 6 hours |
| Combining meshes | 2 minutes | 5 minutes |
| Decimation | 2 minutes | 6 minutes |
| Final triangle count | 519.000 | 1.800.000 |

[2]https://theia.io/optim/

### 2.2.2 Automatic UV mapping

In section 2.1.8, it was referenced that most parameterization algorithms can only handle surfaces homeomorphic to a disk.

Since CAD model parts are closed surfaces, it is necessary to introduce seams when passing the mesh into a 2D domain representation. This causes the process of UV mapping to have two parts: the computation of optimal cuts, as so to reduce seam length and discontinuity artifacts, and the minimization of distortion of triangles of the mesh mapped onto the plane, as so to reduce distortion artifacts [PTH+17]. The more seams are introduced the lowest the obtained distortion will be, but too many seams will result in a poor parameterization with too much discontinuity [SCOGL02].

The methods introduced in this section aim at optimizing distortion and cuts simultaneously, through different methods.

#### 2.2.2.1 Bounded-distortion Piecewise Mesh Parameterization

The algorithm proposed by [SCOGL02] starts off with a random seed triangle. This triangle is mapped to the plane without any distortion, and constitutes the initial patch, with its edges being referred to as the patch front. The algorithm then examines all of the triangles adjacent to the patch front, grading each of the "free" vertices of these triangles (vertices that don't already belong to the patch) according to multiple criteria such as distortion of the resulting flattened triangle. New triangles are then added to the patch iteratively, by selecting the vertex with the highest grade and mapping it to the plane, with its "free" neighbours' grades being recalculated based on its position. This mapping has a distortion threshold defined by the user and is checked for intersections with the patch, as to avoid overlaps. When there are no more triangles that can be added to the patch due to the previous constraints, a new unmapped triangle is selected randomly to start a new patch, with the algorithm terminating when there are no more unmapped triangles left.

The distortion measure used is given by the Jacobian of the transformation function between the original 3D triangle and its counterpart on the plane, with it treating stretching and shrinking the same way. Other criteria can be added to the distortion measure like crease angles or the ratio between the patch area and its squared perimeter, in order to avoid long, thin patches.

#### 2.2.2.2 Autocuts

Autocuts [PTH+17] tries to parameterize meshes with minimal distortion and minimal length of cuts by optimizing an energy function that takes into account both measures.

The distortion measure is the symmetric Dirichlet energy [SS15], which computes the Frobenius norm (square root of the sum of the absolute squares of elements of a matrix) of the Jacobian of the transformation associated with each face. The measure that defines a seam (separation of an edge into two) is given by a monotonic function that is either 0 if the projection of an edge is coincident in both faces, or 1 if the distance between endpoints is different than zero. These measures are weighted over the area of the faces and length of the edges, respectively, and are

then balanced by a $\lambda$ value defined by the user.

$$\min_X E(X) = \min_X (1-\lambda) \sum_{facek} A_k (\|J_k(f_k(X))\|_F^2 + \|J_k^{-1}(f_k(X))\|_F^2) + \lambda \sum_{\{i,k_1,k_2\}\sim j} l_j s(\|x_{ik_1} - x_{ik_2}\|) \tag{2.2}$$

The previous function is non-smooth and non-convex, which means it is difficult and computationally expensive to minimize directly. To solve this problem Autocuts use an homotopy optimization technique that uses a $\delta$ value to control the smoothness of the function. The initial smoothed function eliminates many local minima, making finding a global minimum easier. In each iteration, the function is sharpened back by reducing the value of $\delta$, and a local minimum of the new smoothed function is found by using the previous minimum as a starting point. This way, this method quickly converges into an optimal solution.

Besides the unassisted cutting and parameterization of the mesh, Autocuts also provides the user the possibility to interact with how the UV mapping is done. This can be achieved by tuning the values of the parameters $\delta$ and $\lambda$, or by bounding the UV shape to a certain rectangle. The user can also paint the mesh so the algorithm avoids that area when cutting, and drag vertices and areas, deforming, tearing and attaching UV islands.

The main limitation of Autocuts is that it doesn't guarantee that no global overlaps exist, delegating that process to the manual interaction by the user.

#### 2.2.2.3 OptCuts

OptCuts [LKK+18] is an effort to improve on the solution proposed by Autocuts.

The method introduced by this paper removes the need for the user to set a $\lambda$ value to balance face distortion and seam length, instead needing a user provided distortion bound, for which it minimizes seam length. The algorithm proposed also has the option to apply bijectivity constraints to the mapping, removing overlaps.

OptCuts starts with a bijective UV map created by Tutte's method (2.1.8), cutting it enough to induce disk topology if necessary. It then alternates between minimizing distortion and seam length, updating the $\lambda$ value iteratively until the function converges to the minimum seam length for the distortion bound.

# Chapter 3

# Methodology

As stated previously, Abyssal as a company specializes in the development of 3D visualization, simulation and digitization capabilities for subsea operations, using Unreal Engine.

A crucial step of Abyssal's workflow is the creation of the assets to replicate rigs and ROVs digitally. Currently, this process is done by manually modelling the required objects based on their respective CAD models. 3D modelling software provides the designer a finer control over how tessellation is done, like controlling polygon density in different parts of the mesh, or removing unimportant features. As a result, very high quality assets can be produced, while also having good performance when used in the game engine.

Because of the extended time it takes to model assets manually, Abyssal has to resort to tools like Datasmith when short on time. As referenced before (2.2.1.1), Datasmith provides acceptable results but severely limits the influence the designer can have on the final result.

Consequently, the main objective of this dissertation is to create a functioning prototype that, like Datasmith, allows the importing of CAD models into Unreal Engine, while giving designers more control over the characteristics on the final model.

## 3.1 Requirements

The main features that were identified in this project were:

- **FE-1** - Import CAD model.
- **FE-2** - Visualize and interact with imported model.
- **FE-3** - Tessellate parts of model.
- **FE-4** - Generate automatic UV mappings.
- **FE-5** - Export to 3D model file.

FE-1, FE-2, FE-3 and FE-5 represent a more technological and practical part of the project with the focal point being the creation of the architecture of the application, the integration of external libraries and the development of the features the user needs.

FE-4 requires a more investigative and innovation focused approach, as its objective is to create a method that not only generates UV coordinates automatically but does so in a way that resembles the way a designer would unwrap the object.

### 3.1.1 Functional requirements

#### 3.1.1.1 FR-1 - Read STEP file

Users should be able to select a CAD file and import it into the application. The preference would be to allow the importing of multiple formats, but the support for STEP files is enough. The importing should generate the same geometry, materials and hierarchy as the original CAD model.

Priority: high.

#### 3.1.1.2 FR-2 - Display CAD model

After import, users should be able to visualize the model.

Priority: high.

#### 3.1.1.3 FR-3 - Control scene

The generated scene should provide users with the standard controls of rotating, panning and zooming, using the mouse.

Priority: high.

#### 3.1.1.4 FR-4 - List parts of scene

There should be a list that shows the parts that constitute the CAD model, identified by their names, and that reflects their underlying hierarchy.

Priority: high.

#### 3.1.1.5 FR-5 - Highlight parts

The color of the parts of the scene should change when the user hovers the mouse over them.

Priority: high.

#### 3.1.1.6 FR-6 - Focus on parts

The user should be able to focus on individual parts of the model by clicking on them. The focused part should become the only visible part in the scene and the camera should center on it.

Priority: high.

#### 3.1.1.7 FR-7 - Show scene information

Information such as the number of parts of the model, and the number of faces and vertices used should be displayed both for the entire scene and for individual focused-in parts.

Priority: medium.

#### 3.1.1.8 FR-8 - Global re-tessellation

The user should be able to control a variable that defines how many faces are created when tessellating the entire object.

Priority: high.

#### 3.1.1.9 FR-9 - Part re-tessellation

The user should have finer control when tessellating, with the application allowing for different parts to have different chord tolerance values, and thus different density of triangles.

Priority: high.

#### 3.1.1.10 FR-10 - Part type re-tessellation

The application should provide a way to select and re-tessellate parts that present similar geometry. For example, all of the screws of the model.

Priority: medium.

#### 3.1.1.11 FR-11 - Show wireframe

The user should be able to toggle between showing and hiding the wireframe of objects. Edges of meshes display a different color, while preserving the material of the part, allowing the user to better visualize the obtained tessellation.

Priority: medium.

#### 3.1.1.12 FR-12 - Show normals

The user should be able to toggle between showing and hiding the representation of the normals of the object. The normals should be displayed as straight short lines at each vertex of the mesh, in a color that contrasts with the surrounding scene.

Priority: medium.

#### 3.1.1.13 FR-13 - Show textures

The application should provide a secondary scene that displays the texture atlas corresponding to the UVs of a part, allowing the user to easily visualize and analyze the generated texture coordinates.

Priority: medium.

#### 3.1.1.14 FR-14 - Automatic UV generation

The application should generate texture coordinates automatically, or semi-automatically, for the parts of the model.

Priority: medium.

#### 3.1.1.15 FR-15 - Export FBX file

The application should be able to create an FBX file with the same geometry, materials and hierarchy of the current scene. Tessellation, normals and UV coordinates should be preserved.

Priority: high.

#### 3.1.1.16 FR-16 - Change grouping of parts

The user should be able to create new groups of parts, and change their composition. Priority: low.

#### 3.1.1.17 FR-17 - Merge parts

The user should be able to select multiple parts and merge their geometry, resulting in a single part. Priority: low.

#### 3.1.1.18 FR-18 - Part separation

The user should be able to cut parts of the model by their edges, generating multiple parts from the same one, which could be useful when parts from the original model are merged together. Priority: low.

#### 3.1.1.19 FR-19 - Select and re-tessellate parts by material

The user should be able to highlight, focus and re-tessellate parts that have the same material.

Priority: low.

#### 3.1.1.20 FR-20 - Delete unseen parts

The application should provide the option to delete unseen geometry, only preserving faces and vertices that constitute the outer boundary of the object. This could greatly reduce the amount of geometry generated, increasing performance of the model, but would also limit what could be done with the object and its parts when in the game engine.

Priority: low.

#### 3.1.1.21 FR-21 - Menu for hotkey assignment

The application should provide the user with the option to assign their own preferred keyboard shortcuts to the different functions available.

Priority: low.

### 3.1.2 Non-functional requirements

- **NR-1** - Import time of CAD models should be equivalent to Datasmith.
- **NR-2** - Controls should be easy to use and understand.
- **NR-3** - User should get feedback on every action.
- **NR-4** - Interaction with the imported part should be seamless and not present any serious lag.

## 3.2 Workflow

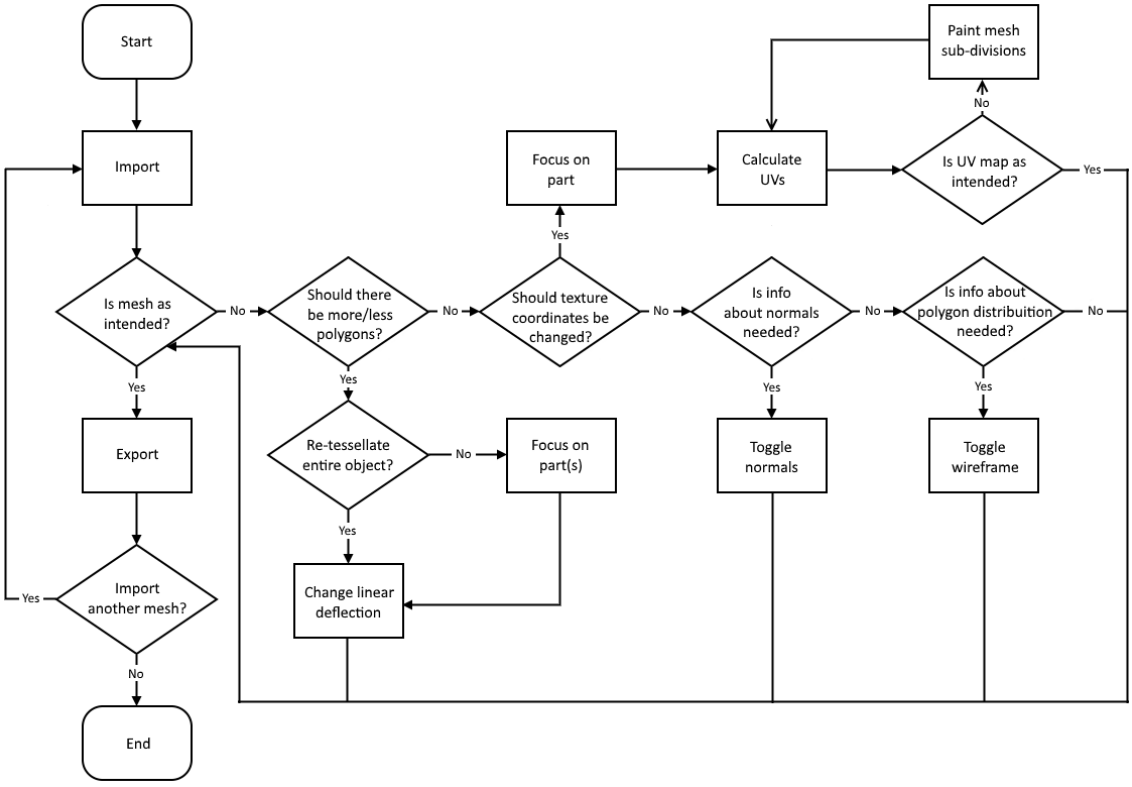


Figure 3.1: Expected workflow of the developed solution

The workflow of the application, as described in 3.1, starts with the importing of a CAD file. A user should be able to select the STEP file of a model to convert and the application should load the file, tessellate the model and display it on the user interface.

After the model is visible the user should then be able to interact with the model, select parts, view their tessellation, normals and additional information.

At this point, the user should be able to re-tessellate the model, or parts of it, if wanted, being possible to both increase or decrease the number of polygon used to represent a surface. It should

also be possible to generate texture coordinates automatically for a selected part, with the user being able to affect how this mapping is done.

Finally, the application should output an FBX file that contains the hierarchy of the parts of the object, its materials, and the geometry of the parts, as updated by the user in the application.

## 3.3 Method and algorithms

While most of the requirements of the application didn't require much research other than reading documentation, two particular features required more in-depth research and theoretical knowledge: the simplification of the geometry of the parts of the model, and the automatic generation of UV coordinates for these parts.

### 3.3.1 Simplification of geometry

The initial approach taken regarding the simplification of geometry was to select and implement a mesh decimation algorithm 2.1.5. These algorithms can provide quick reduction of the number of vertices and faces used to represent a surface by using various methods, both local and global.

Despite their good performance timewise, the results obtained when using these algorithms weren't satisfactory visually, due to the limitation discussed in section 2.1.5.4.

Because of the problems introduced by the mesh decimation algorithms, another approach had to be taken, with the method selected being the re-tessellation of the selected part. This method, despite being slower, obtained the best results. By re-tessellating, new vertices are sampled along the surface, which allows the resulting mesh to better approximate the original surface, while also maintaining flow associated with it. A comparison between the re-tessellation method and the previously introduced mesh simplification algorithms (2.4) can be seen in 3.2.

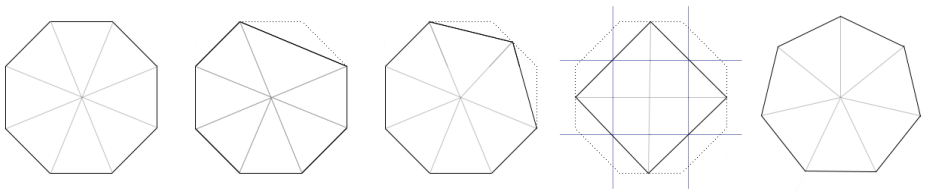

Figure 3.2: Comparison between mesh simplification algorithms and re-tessellation

### 3.3.2 Automatic UV mapping

As seen in section 2.2.2, the simultaneous optimization of cuts and distortion in UV mapping is a complex problem. What these algorithms try to achieve is to minimize objective metrics such as seam length and distortion. In practice, this approach often results in "ideal" seams that don't look as good as the seams defined by a human designer.

Instead of handling the cutting and parameterization of the mesh at the same time, the developed method intended to emulate how designers map UVs: first determine where is the best place to introduce seams, then calculate the texture coordinates.

#### 3.3.2.1 Step 1 - Create sub-division of the mesh

The first step of the developed method is dividing the mesh into sub-divisions, or patches. Since the CAD models represent parts that had to be manufactured and assembled, it's normal for there to be straight angles, in order for everything to fit together. These natural seams are a good starting point to begin the patching process, as the discontinuity of the mapping becomes less noticeable when the cuts are made alongside these seams.

In the triangular mesh, this criteria for patching means dividing the surface alongside the edges whose vertices have multiple normals pointing in different directions (normal discontinuity).

#### 3.3.2.2 Step 2 - Mesh preparation

One of the advantages of step 1 is that the patches obtained either have disk topology or close to it. This is especially useful as it allows the application of surface parameterization algorithms directly, like those mentioned in section 2.1.8.

The patches that aren't homeomorphic to a disk can generally be categorized into three groups: surfaces with holes, open surfaces of revolution (cylinders, open cones...), and the combination of the first two. The meshes associated with these patches have to be prepared and turned into meshes with disk topology, in order to apply a parameterization algorithm.

For the surfaces with holes, the approach is to create a vertex at the center of the boundary of each hole. This vertex is then connected to each of the vertices of the boundary, eliminating the hole.

For surfaces of revolution, the solution is to cut the mesh from one boundary to the other. This is done by the following steps:

1. One of the boundaries is identified as stack 0.
2. All of the vertices adjacent to the vertices of stack 0 that aren't in stack 0 constitute stack 1.
3. This is repeated until a stack is formed that has vertices on a boundary.
4. A random vertex from stack 0 is selected and becomes the first vertex of the cut.
5. The orientation of the mesh between stack 0 and stack 1 is given by the difference between the centroid of both stacks.
6. The next vertex of the cut is: contained in stack 1; adjacent to vertex 0; the vertex in which the edge that connects it to vertex 0 makes the smallest angle with the orientation of the mesh.
7. This is repeated until the final stack is reached.

8. The vertices alongside the cut are duplicated and the faces on one side of the cut are edited to use these new vertices, this creating a mesh with disk topology.

This method results in seams that closely follow the orientation of the mesh, resulting in much more regular discontinuities, which look better.

It is important to notice that finding an edge flow on a triangular mesh is only possible if the mesh is tessellated in a regular way, equivalent to a quad mesh. For irregular meshes the method will produce sub-optimal results.

Simultaneous cutting and hole-filling is not supported, which results in unsatisfactory results for surfaces of revolution with holes. This is because, even though it is easy to the determine if a vertex is on a boundary or not, there no reliable, general way to determine if that boundary is a hole or the second outer boundary of the surface.

The ambiguity between what is considered an outer boundary and what is considered a hole also affects the decision of how to categorize a surface.

A practical example of this is given by the comparison between a flat circle with a circular hole in its center (first surface in 3.3) and an open cone (third surface in 3.3). If the axis of both of these surfaces is aligned with the Y axis, and the vertices of the topmost boundary of the open cone are translated to Y=0, two similar surfaces are obtained. Therefore, even though one is an open surface of revolution and the other is a surface with a hole, they are homeomorphic to each other. The problem with this then obviously becomes where alongside the translation does the surface change from one category to the other (second surface in 3.3).

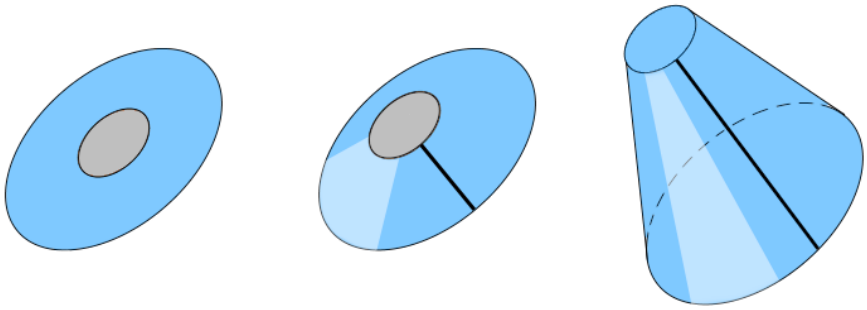

Figure 3.3: Surface 1 gets hole-filled, surface 3 gets cut, what about surface 2?

Answers to this problem could involve some heuristic involving the area of the boundaries, how many vertices they have, or the direction of the normals alongside them. Instead the approach taken was to consider both options as possible for every mesh. Thus, UV mappings are calculated for both approaches, as well as their respective distortion error, with the final mapping being the one that has the lowest error value.

#### 3.3.2.3 Step 3 - Parameterization

The final step of the process it to parameterize the obtained patches.

In this project, the chosen parameterization algorithm was Boundary First Flattening [SC17], both because it is fast and generates a flattening with minimal area distortion and virtually zero angle distortion, and also because the library that implements it was easy to integrate.

The UV coordinates generated by the algorithm become the new UV coordinates of the part.

# Chapter 4

# Implementation

## 4.1 Architecture

Succinctly, the solution developed can be described as an Electron application, using Three.js to handle the 3D graphics and two Python modules, OpenCascade and FBX SDK, for importing and exporting, respectively. The architecture of the application can be seen in diagram 4.1.

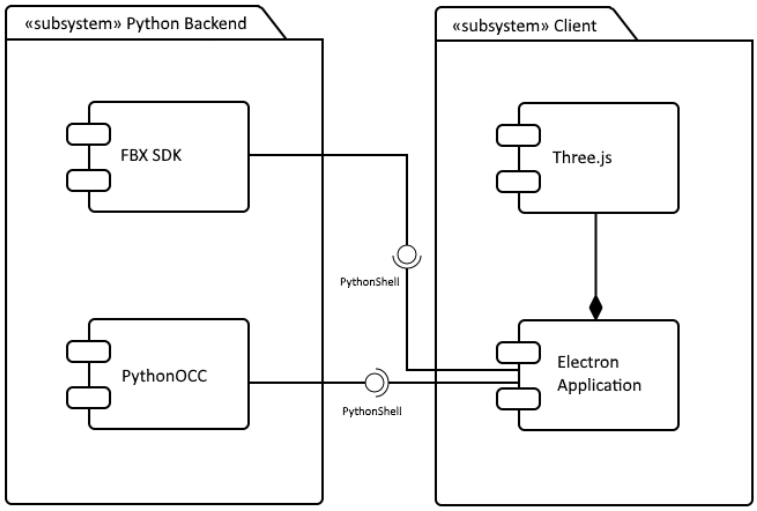


Figure 4.1: Component diagram of the application

### 4.1.1 Electron

Electron is a Javascript framework that allows the development of desktop applications using web technologies: HTML, CSS and Javascript.

An Electron application consists of two type of processes, main and renderer processes.

The main process is unique and its main function is to create and manage web pages and their respective renderer processes. This process can be seen as a variant of Node.js that is focused on desktop applications instead of web servers, thus allowing it to interact with the operating system [Elea].

The renderer processes are each associated with a web page. They are isolated from each other and only manage the HTML, CSS and Javascript associated with their own page. These pages are displayed by using Chromium [Elea].

Main and renderer processes can communicate by Electron's IPC (inter-process communication), which is based on named pipes [Chr]. This is done by sending JSON messages back and forth between the two processes, either synchronously or asynchronously. Renderer processes can also invoke methods of the main process without sending messages by using the remote module. This module exposes RPC (remote procedure call) functionality to global variables defined on the main process, similarly to Java's RMI [Eleb].

There are multiple reasons that lead to the decision of using Electron as the foundation of this project:

- Electron allows for the development of a responsive and user-friendly interface in an easy and fast way;
- It provides access to a large collection of Node.js modules, which speeds up development;
- It supports running executables, which can be created using languages other than Javascript, or scripts like those created using Python;
- Javascript has the WebGL API that allows rendering interactive 2D and 3D graphics;
- Electron applications can be packaged to target different operating systems without much effort.

### 4.1.2 PythonShell

Some functionalities that are fundamental to this project, like importing CAD models, tessellating objects and creating an FBX file from the scene, are not available in Javascript modules. The two options to resolve this issue are to either build these features from scratch, which wasn't a viable option considering the time constraints and amount of work required, or find libraries in other languages that implemented these features and a way to integrate them into the project.

PythonShell is a Node.js module that can spawn Python scripts in a child process. It launches a script by issuing the python command through a command line running in the background, and the communication between this process and the process that launched it is done through standard input and output (stdin/stdout).

By default PythonShell launches a script using the Python installed on the machine of the user. This isn't ideal as the user could lack the required libraries or even Python itself. Luckily this module allows to explicitly define the path of the Python instance to use, being a solution to just ship the application with a basic installation of Python with the required libraries.

### 4.1.3 OpenCascade

Open Cascade is a CAD development framework in C++ that provides features such as topological and geometrical operations, and data exchange (import/export) of commonly used CAD file formats.

PythonOCC is a python library that wraps OpenCascade's open source version (Community Edition) by using SWIG. It requires Python 3.6, and allows the reading of STEP files into an internal representation that can then be exported to JSON and imported in Three.js.

### 4.1.4 Three.js

One of the main focus of this project is the interaction of the user with the tessellation and texture coordinates creation process. As such, having an interactive 3D scene was seen as fundamental.

WebGL is a Javascript API that allows rendering 3D graphics, creating scenes, geometry, materials, applying materials, using shaders, among other functionalities. Despite this, WebGL is also very low level, which means it requires quite a bit of work to accomplish some of these tasks.

Three.js is Javascript library that provides higher level abstractions to creating and rendering 3D graphics, built on top of a WebGL renderer. Three.js provides built-in functionaries such as different types of cameras, lights and controls, easy interaction with the geometry and the scene, allowing for effortless modification of coordinates and visibility, and removing the necessity to directly modify shaders.

Besides these features, one aspect of Three.js that is especially important for this project is its ability to import and export scenes from, and to, different formats.

Since OpenCascade provides the option to export objects to a Three.js compatible JSON string, the interoperability between tessellator and renderer only required a few modifications.

For the exporting of the scene, the final target file format was FBX, as that is the only format that can be imported to Unreal Engine directly that preserves hierarchy and materials 2.1.10. Three.js doesn't provide an exporter to FBX but it does have a COLLADA exporter, a file format that also stores the hierarchy and material information, and that can be converted to FBX through the FBX SDK introduced next.

### 4.1.5 FBX SDK

Autodesk's FBX SDK (software development kit) provides official support to editing and converting FBX files.

This SDK provides methods in C++ and their respective bindings in Python, requiring Python 3.3. Since this SDK and PythonOCC need different Python versions, the solution found, albeit a sub-optimal one, was to include installations of both versions in the project directory. PythonShell then invokes a different version of Python according to what script it is running.

## 4.2 User interface

The application is designed in a single page, using HTML, CSS and Javascript. Three.js is used for the canvases and jQuery is used to handle DOM interaction and events. Materialize.js is used for the icons used as well as to show tooltips when hovering over each button. These tooltips give information about the function of each button and the keyboard shortcut associated with its function.

The elements in the main menu bar have the following functions (in order):

- Import
- Export
- Increase linear deflection
- Change lines deflection
- Decrease linear deflection
- Toggle normals
- Toggle wireframe
- Toggle textures
- Divide by patches
- Calculate UVs

The main container of the application, as seen in 4.2, consists of four elements: part lister, main canvas, UV canvas, and scene stats.

- **Part lister** - on the left, shows the name of all of the parts of the imported object, as well as their hierarchy. Groups of parts can be collapsed and expanded by clicking on the arrow to their right;
- **Main canvas** - in the middle, shows the imported object or selected part in 3D. It allows the user to rotate, pan and zoom the scene;
- **UV canvas** - on the right, allows the user the visualize the texture coordinates of a selected part. This canvas can't be interacted with;
- **Scene stats** - on the bottom, shows how many parts are currently visible, as well as how many vertices and polygons are being used to represent them.

The components of the main container can be resized width wise by clicking and dragging the vertical black bars (to the right of the scroll bar of the part lister and in the center).

In order to make the application more visually consistent, a frameless window is used. This removes the default OS frame and menu, but required manual creation of the minimize, maximize and close buttons and their functionality.

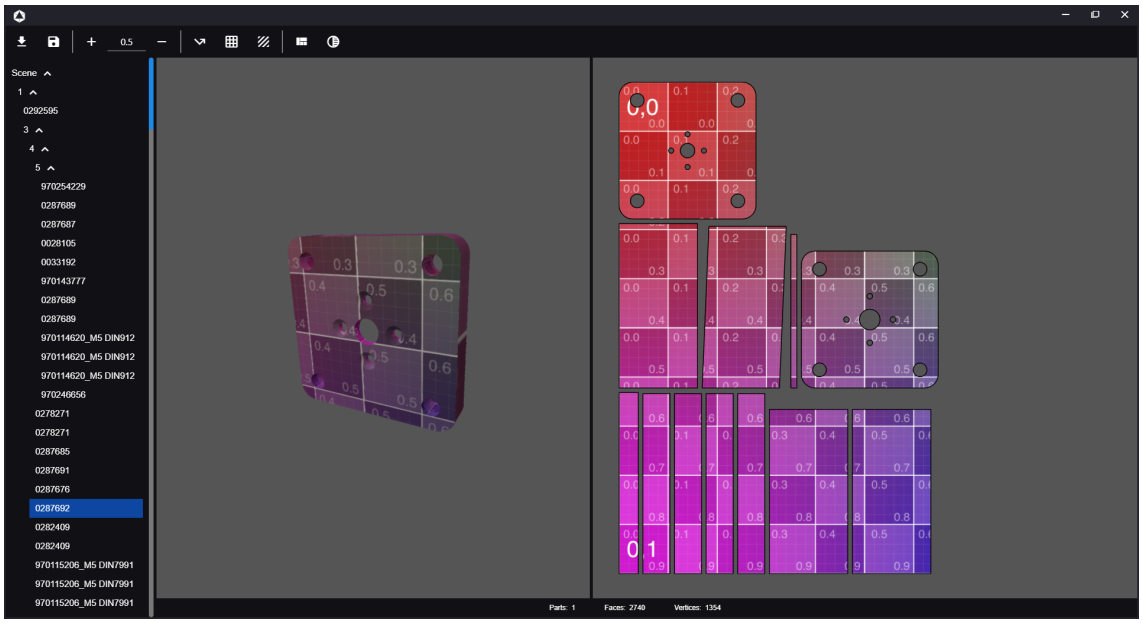


Figure 4.2: Screenshot of the application showing a part of the object and its UV map

## 4.3 Use cases

### 4.3.1 Import file

| | |
|---|---|
| Preconditions | • Application loaded |
| Basic flow | 1. Click on import button or press I key.<br>2. The system opens a file selection window.<br>3. The user navigates to the intended directory and selects the STEP file to import. |
| Postconditions | • A loader is presented on the screen while the file is imported.<br>• The elements of the scene are cleared.<br>• The parts are progressively shown on the canvas when loaded.<br>• The camera position is updated to show every part of the model.<br>• The part list is updated with the names of the parts of the model and their hierarchy. |

### 4.3.2 Export scene

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Click on export button or press E key.<br>2. The system opens a file save window.<br>3. The user navigates to the intended directory and inputs name for the FBX file to be saved as. |
| Postconditions | • The file is created. |

### 4.3.3 Highlight part

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Put mouse over part on main canvas or over name of part on the part lister. |
| Postconditions | • If the part is visible, it is outlined and its color is changed to blue on the main canvas.<br>• The background of the name of the part on the part lister changes to blue. |

### 4.3.4 Highlight parts by type

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Put mouse over part on main canvas or over name of part on the part lister, while pressing shift. |
| Postconditions | • All of the parts that have the same shape and size as the selected part are outlined and have their color changed to blue on the main canvas.<br>• The background of the name of each of these parts on the part lister changes to blue. |

### 4.3.5 Highlight parts by material

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Put mouse over part on main canvas or over name of part on the part lister, while pressing alt. |
| Postconditions | • All of the parts that have the same material color as the selected part are outlined and have their color changed to blue on the main canvas.<br>• The background of the name of each of these parts on the part lister changes to blue. |

### 4.3.6 Focus on part(s)

| | |
|---|---|
| Preconditions | • At least one highlighted part |
| Basic flow | 1. Left click on a part on the main canvas or on its name on the part lister (a part is already implicitly highlighted before it's clicked on). |
| Postconditions | • All of the highlighted parts become visible, while the rest of the parts are hidden.<br>• Shows the texture atlas of the selected part, if only one part is focused on, it has calculated texture coordinates, and the UV canvas is open.<br>• The camera focuses on the selected parts. |

### 4.3.7 Unfocus from part

| | |
|---|---|
| Preconditions | • Being focused on a part or multiple parts |
| Basic flow | 1. Right click anywhere on the main canvas. |
| Postconditions | • All of the parts of the object become visible.<br>• Clears the UV canvas. |

### 4.3.8 Re-tessellate part(s)

| | |
|---|---|
| Preconditions | • At least one visible part |
| Basic flow | 1a. Click on the increase/decrease linear deflection buttons or press the +/- keys.<br>1b. Or input the value for the new linear deflection on the input field, and focus out of it. |
| Postconditions | • A loader is presented on the screen while the parts are tessellated.<br>• The re-tessellated parts have a different number of polygons and vertices. |

### 4.3.9 Toggle normals

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Click on the toggle normals button or press the N key. |
| Postconditions | • A representation of the normal vectors of each visible vertex is shown.<br>• Focusing on parts and unfocusing updates the normals that are visible.<br>• If the normals were already visible, they are hidden. |

### 4.3.10 Toggle wireframe

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1. Click on the toggle wireframe button or press the W key. |
| Postconditions | • Lines are displayed that show the edges of the visible parts.<br>• Focusing on parts and unfocusing updates the lines that are visible.<br>• If the wireframes were already visible, they are hidden. |

### 4.3.11 Toggle textures

| | |
|---|---|
| Preconditions | • Loaded model |
| Basic flow | 1a. Click on the toggle texture button or press the T key.<br>1b. Or drag the right most resize bar horizontally. |
| Postconditions | • A standard colored grid texture is applied to the visible parts.<br>• The UV canvas is opened.<br>• If only one part is visible and it has calculated texture coordinates, its texture atlas is shown in the UV canvas.<br>• Parts have the texture applied while the UV canvas is shown, even through focusing and unfocusing.<br>• If the UV canvas was already open, the textures are hidden and the UV canvas closed. |

### 4.3.12 Divide by patches

| | |
|---|---|
| Preconditions | • Exactly one visible part |
| Basic flow | 1. Click on the divide by patches button or press the P key. |
| Postconditions | • If the UV canvas is not open, it is opened.<br>• The mesh of the part is divided by normals discontinuity and the resulting patches are colored in different random colors.<br>• Clicking on the button or pressing the P key again resets the patches and recolors them. |

### 4.3.13 Paint patches

| | |
|---|---|
| Preconditions | • Exactly one visible part<br>• Patching being visible through 4.3.12 |
| Basic flow | 1. Left click on a patch, on the main canvas, while pressing the Ctrl key, and drag over adjacent patches. |
| Postconditions | • Recolors the adjacent patches to the color of the first clicked patch, merging the subdivisions of the mesh. |

### 4.3.14 Calculate UVs

| | |
|---|---|
| Preconditions | • Exactly one visible part |
| Basic flow | 1. Click on the calculate UVs button or press the Space key. |
| Postconditions | • A loader is presented on the screen while the UVs are computed.<br>• Calculates texture coordinates for the visible part.<br>• Uses the default division of the mesh, or the one altered by the user in 4.3.13<br>• The texture applied to the part reflects the new UV coordinates.<br>• The texture atlas corresponding to the UV coordinates is shown on the UV canvas.<br>• Calculates texture coordinates for the visible part. |

## 4.4 Component specification and implementation

Disregarding configuration files and libraries, the structure of the project is as follows:

```
|--main.js
|--index.html
|--buffering.html
|--style.css
|--src
|    |--components
|    |    |--Canvas.js
|    |    |--FrameHeader.js
|    |    |--PartLister.js
|    |    |--SceneStats.js
|    |    |--UVCanvas.js
|    |--events
|    |    |--EventHandler.js
|    |--features
|    |    |--Exporter.py
|    |    |--Highlighter.js
|    |    |--Importer.py
|    |    |--NormalsHelper.js
|    |    |--PartFocus.js
|    |    |--PartTypes.js
|    |    |--Tessellator.py
|    |    |--UVCalculator.js
|    |    |--WireframeHelper.js
|    |--utils
|    |    |--HalfedgeGeometry.js
```

The main.js file is the entry point of the application. It creates the main window of the application using the index.html file, and a secondary window for the buffering animation using buffering.html.

In index.html the structure of the application is defined, jQuery and Materialize are imported and the components that handle the application (src/components) are created. Each of the components are defined as singletons, and declaring them in the HTML file makes them global variables, facilitating interaction between components.

The buffering animation is defined in buffering.html. This file is used on the secondary window of the application, which is opened when asynchronous operations are executed, in order to give visual feedback to the user's actions, closing when they are concluded. This window is a modal window, which disables the main window of the application while it's active. It is transparent, showing only the buffering animation, and is always centered according to the main window.

All of the style of the application is contained in style.css.

### 4.4.1 Import file

When the user selects the file to import, the full path for the file is obtained, which is then passed on to Importer.py.

In Importer.py, the file location is used in the file reader provided by PythonOCC. This reader starts by reading the entire file and creating a graph representation of it, also referred to as document. Different classes are available to interact with this document that focus on different aspects of the model: to interact with the shapes of the model a Shape Tool should be used, to obtain colors a Color Tool should be used. Through the Shape Tool it is possible to obtain if a shape is a compound (group or assembly of shapes) or a simple shape.

The graph is then traversed from its root, a compound shape, down to its leaves, which are simple shapes that have a geometry associated with them. While traversing the graph, the attributes of each shape that are necessary for Three.js are sent through the standard output, formatted in JSON. For compound shapes, their id, name and id of their parents are sent. For simple shapes, the information about their color, geometry and linear deflection used is added.

It is at this step of traversing the graph that the tessellation process is done. When a simple shape is reached, it is passed on to the tessellator provided by PythonOCC, along with a default linear deflection value. The tessellator then outputs arrays of vertex coordinates, normals and UVs that are compatible with the representation used by Three.js.

The re-tessellation method provided by this application had the objective of being responsive and fast, to try and improve the lengthy process of plugins like Datasmith and, thus, increase the productivity of the design team. Because of this, it was simply not feasible to have to re-read the STEP file on re-tessellation. The options were to either keep the Python process running and have the graph structure stay in memory, send the PythonOCC representation of the surface (TopoDS) inside the JSON to the main process, or store these representations in temporary files.

Keeping the Python process alive required the constant need to poll for a re-tessellation request, which would not happen often enough to justify it. It also required keeping the model graph

in memory, which could be heavy performance wise. On the plus side, it would be faster than the other two methods since it wouldn't rely to data transferring.

Serializing the TopoDS objects and sending them to the main process also kept them in memory, but allowed for the re-tessellation process to be executed only when needed. The main problem that arouse with this option was the fact that the serialization of TopoDS objects produced very large strings, which could cause problems when reading and writing to stdio, when using large, complex models.

Finally, the used option was to store TopoDS serializations in temporary files. This decision allowed to reduce how much memory was used, which is important for more complex models, while not increasing load time significantly compared to the previous method. Temp files are deleted when the application is closed.

### 4.4.2 Scene creation

As scene parts are read from the STEP file, they are added to the scene. This implies creating a geometry and material for each one, checking and removing invalid triangles (since the tessellator can create zero area triangles), and updating the camera location to keep every part visible.

The geometry of the parts is defined by Three.js' BufferGeometry, which represents vertices, normals and UVs as typed arrays of floats, with each element corresponding to a coordinate. Since there is no information on the STEP file about reflectivity and emissivity of the materials used, the materials used in the scene are standard Three.js matcap (material captures) materials, which statically define lightning and reflectivity in the material. This also closely resembles the imported object in Unreal Engine.

Every time a part is added to the scene, an event is triggered in EventHandler.js that adds the name of the part to the part lister using jQuery, and updates the scene stats, by adding the number of vertices and polygons of the new part to a counter.

When the scene finishes loading, a map is created that holds information about part types. Parts that have the same type are stored together, with a type being defined by the number of vertices in the mesh and the radius of its bounding sphere. This method is used since it is fast and produces acceptable results (example: a copy of a screw that is translated to another position and rotated will maintain the same geometry and size), while not requiring complicated geometry comparison operations. Setting types on import enables finding parts of the same type as a selected part in constant time, thereby increasing performance.

### 4.4.3 Highlight part

When hovering the mouse on the main canvas a raycast is done following the orientation of the camera. The first part of the model this raycast intersects is the part to highlight. A part can also be highlighted when hovering on the part lister, since each of the HTML elements that are contained in it have the id of the part they correspond to.

If the user intends to select all parts of the same type or material as the highlighted part, these are obtained from the data structures in PartTypes.js. All of the parts to be highlighted are stored in an array in Highlighter.js, have their material changed to blue (the original color is saved in order to unhighlight it correctly) and have a shader applied to them, displaying their outline in a different color.

### 4.4.4 Focus on parts

To focus on a part, or parts, PartFocus.js receives the array of highlighted parts of Highlighter.js. These parts become visible, if they weren't already, and all other parts become invisible.

To click on a part means that the mouse has to be over it, which in turn signifies it has to be highlighted, making this a valid approach.

After the part, or parts, are focused on, the bounding box of the visible objects is calculated and the camera is re-positioned and oriented to fit all of the parts.

### 4.4.5 Normals and wireframe helpers

Three.js already provides the functionality of creating normal helpers and geometry that only presents edges. For each of the parts of the scene, when they are created, the corresponding helpers are also created.

Toggling normals and wireframes simply changes the visibility of these elements in the Three.js scene. On re-tessellation these elements are modified to reflect the new geometry.

Since there is one normals helper and one wireframe helper per part of the object, when they become visible, in scenes with a lot of parts, it noticeably affects performance as each of them effectively doubles the geometry of the scene that has to be rendered.

### 4.4.6 Re-tessellation

Each of the simple shapes existent in the scene have associated with them a tessellation quality, the linear deflection of their mesh, and a temporary file that describes their underlying surface in PythonOCC classes, as stated in 4.4.1.

When a part is re-tessellated, a new Python process is started using Tessellator.py, and the id of the part and its new linear deflection value is sent to this process via standard input. The file associated to the part's shape is then read, and its contents are decoded and tessellated similarly to the import process.

Finally, the old geometry of the part is replaced with the output of the tessellation, and the normals and wireframe helper of the part are updated.

When multiple parts are re-tessellated at the same time an array is sent with the information necessary of all of the parts, and each of them is tessellated and updated on the canvas iteratively, similarly to the import process.

### 4.4.7 UV coordinates calculation

The foundation of the developed method to calculate UVs automatically is the Boundary First Flattening algorithm implemented in the library geometry-processing-js [1]. The focus of the work done was how to integrate this library into the project and adapt it so non-disk topology objects could have their UVs mapped.

To flatten a mesh, the BFF algorithm that was used requires two data structures, a polygon soup, which contains the coordinates of all vertices of the mesh, and a halfedge mesh data structure.

A halfedge mesh is a structure that stores the connectivity of a mesh by dividing each of its edges into two halfedges. Halfedges can be defined as the instance of an edge in a face. The two halfedges of an edge have their origin in the two opposing vertices of that edge and belong to different faces. A halfedge stores information about its face, edge and vertex, as well as references to the halfedges that belong to the same face as itself and the other halfedge of its edge. Because of this, from a single halfedge the entire mesh structure can be navigated. An example of the structure can be found in appendix B, and can be visualized in 4.3.

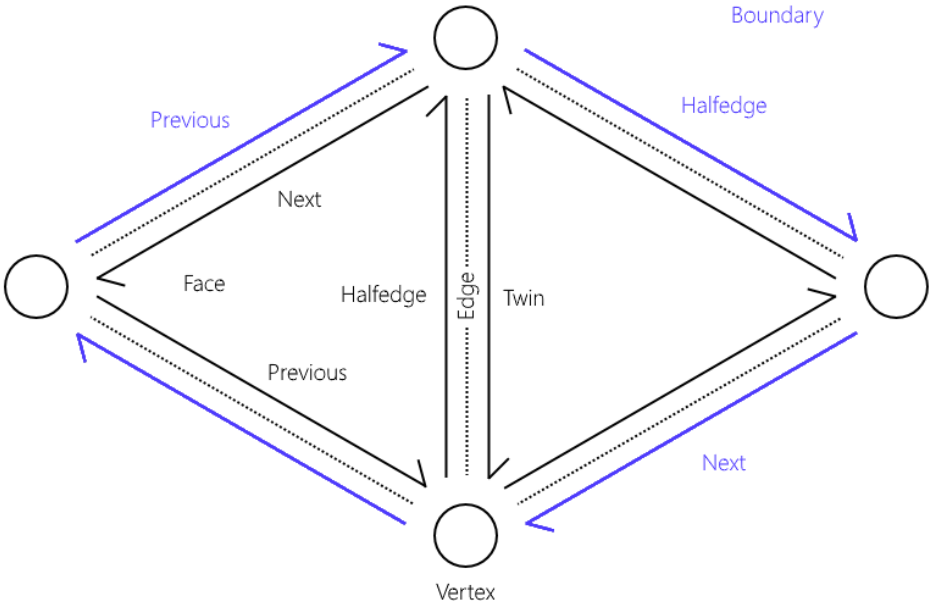


Figure 4.3: Example of a halfedge mesh structure

The halfedge mesh structure provided by geometry-processing-js can be constructed by passing on to it arrays of vertex coordinates and face indices.

[1] https://github.com/GeometryCollective/geometry-processing-js

The original mesh of the object is first divided by the points that have multiple normal vectors with different orientations. This is done by starting a new subset with a random triangle, and then iteratively adding adjacent triangles to it. If the vertices of the common edge between a triangle of the subset and an un-visited triangle have different face normals, then this un-visited triangle is not added to the subset. New subsets are created until every polygon of the mesh belongs to a subset.

When the user paints patches on the main canvas, what is done is a simple merge of the list of points of both subsets into a single subset.

Calculating the UVs of a part entails iterating over every subset of the mesh creating a halfedge mesh structure for each of them, applying the BFF algorithm and packing the obtained UVs into the 0 to 1 range, without overlapping.

The application of the BFF algorithm can be done directly if the subset has disk topology or it requires extra preparation as stated in 3.3.2.

The process of hole-filling consists of, first, calculating the longest boundary of the subset. This is done by iterating over each of the edges of every boundary and summing up their length. For every boundary that is not the longest a vertex is created in the polygon soup whose coordinates are the centroid of the polygon created by the boundary's vertices. The halfedge geometry is also altered, with this new vertex being added and halfedges created between it and every vertex of the boundary, thereby creating new faces that eliminate the hole.

The process of cutting the mesh requires finding its flow. All of the vertices adjacent to one boundary are found, with these vertices constituting a stack. This process is repeated until a stack has a least one vertex on a boundary. For each of the stacks its centroid is calculated, with the difference between one centroid and the next being the orientation of the mesh between those two stacks. Then, one of the vertices of the first boundary is selected randomly, and the orientation of the plane created by this vertex, the centroid of the boundary and the centroid of the next stack is given by the cross product of the vectors centroid2-centroid1 and vertex-centroid1. For every vertex adjacent to this one on the next stack the same is done, with the next vertex selected being the one whose cross vector makes the smallest angle to the previous cross vector.

When a vertex is obtained in each of the stacks, the cutting process starts. This is done by basically duplicating each of the vertices alongside the cut in the polygon soup, changing the faces on one side of the cut, on the halfedge mesh, to use these new vertices, and updating the connectivity of the halfedges, thus creating one continuous outer boundary.

Both the hole-filling process and the cutting process are done for each non-disk topology subset, with the BFF algorithm being applied to the two sets of polygon soups and halfedge meshes. To decide what is the best method for the given part, the quasiconformal error is calculated for both mappings. This method is also provided by geometry-processing-js and calculates how much the internal angles of the faces of the parameterized surface differ from the angles of their counterparts in 3D. The UV coordinates of the mapping with the lowest error become the final UVs of the part.

```
Data: Halfedge geometry
Result: UV coordinates array
// Divide by normals
while not all faces mapped do
    start new subset with random unmapped face;
    for faces adjacent to patch do
        if vertex normals in face == vertex normals in patch then
            add face to patch;
        end
    end
end
for subdivision of subdivisions do
    // Hole fill
    get longest boundary;
    for boundary not longest do
        create vertex on centroid of boundary vertices;
        remake connections;
    end
    UV1 ⟵ apply BFF to mesh;
    // Cut mesh
    first stack ⟵ boundary vertices;
    while vertex of new stack not on boundary do
        new stack ⟵ vertices adjacent to previous stack;
    end
    seam start ⟵ random vertex of first stack;
    while stack != last stack do
        for vertex adjacent to seam vertex do
            angle ⟵ angle(adj_vertex-seam_vertex, stack_center-prev_stack_center);
        end
        seam vertex ⟵ vertex(angle == min(angle))
    end
    remake connections;
    UV2 ⟵ apply BFF to mesh;
    if distortion(UV1) < distortion(UV2) then
        final UV ⟵ UV1
    else
        final UV ⟵ UV2
    end
end
// Pack to [0, 1] space
get bounding box of all UVs;
pack using guillotine method;
divide UVs by container size;
translate UVs by box location;
```

**Algorithm 1:** UV calculation algorithm

The final process of the generation of the UV coordinates is packing the obtained UV coordinates into the 0 to 1 UV space, so a single texture file can be used when applying textures to the object.

The packing of the texture coordinates starts by calculating the bounding rectangle of each patch, by taking the U and V coordinates of each point and calculating their maximum and minimum. On top of this, a constant padding is applied to every bounding rectangle to avoid texture bleeding (2.6). After this, a bin-packing algorithm is applied based on the solution found here[2]. This method is a guillotine method (2.1.9.2) that always divides the space horizontally, which produces quite acceptable results at good performance, despite being a very simple method. This algorithm also sorts the rectangles by height before packing, which optimizes vertical space. The UVs are then translated by the coordinates of their corresponding packed rectangle, and all of these values are normalized to values from 0 to 1, by dividing by the size of the packing container.

[2] https://observablehq.com/@mourner/simple-rectangle-packing

# Chapter 5

# Results

This chapter presents the results obtained when using the developed application, as well as its evaluation by the designers and developers at Abyssal.

The main focus of the tests done was to compare CADto3D with Datasmith to understand if the former was a viable alternative to the latter. To do this, the performance of the workflow of both applications was measured and the 3D models created were evaluated for visual fidelity and geometry complexity.

This chapter also presents an overview of the results obtained with the UV calculation method proposed, with visual representation of these available in appendix C.

## 5.1 Results and workflow comparison with Datasmith

The main workflow of the application developed is the import of a STEP file and subsequent export of an FBX. To evaluate the performance of this process, the two models of section C.1 were used, and the time taken by this process was compared to the import time of Datasmith on Unreal Engine.

When tested, the obtained duration of each process were as seen in 5.1.

Table 5.1: Comparison between import times using Datasmith and the developed application using linear deflection of 1

| | Object1 | | Object2 | |
|---|---|---|---|---|
| | Datasmith | CADto3D | Datasmith | CADto3D |
| Import | 70s | 75s | | 340s |
| Export | | 5s | | 25s |
| Import into Unreal | | 30s | | 160s |
| Total import time | 70s | 110s | failed | 525s |
| Re-tessellation | 60s | 30s | | 130s |

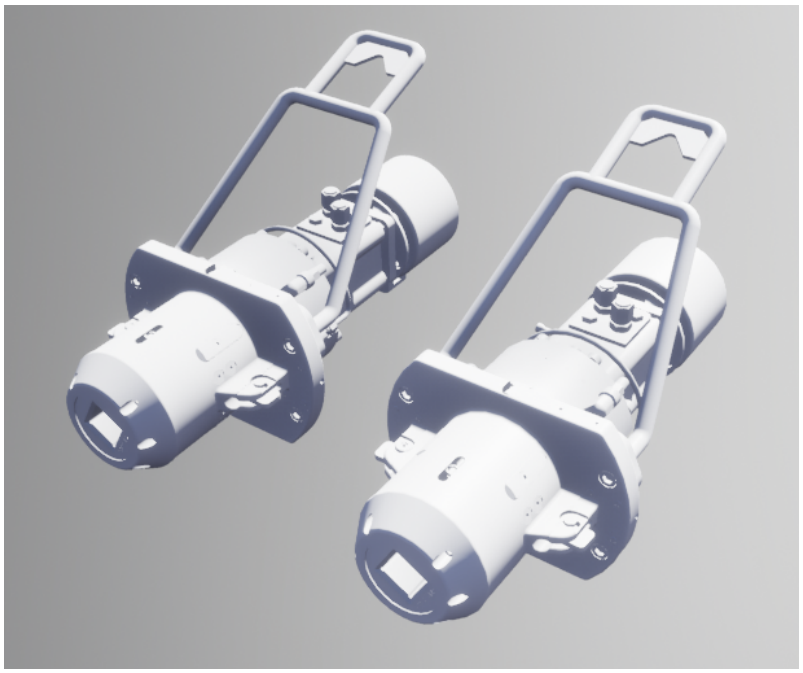

Figure 5.1: Comparison between models obtained with Datasmith (left) and CADto3D (right) in Unreal Engine. The developed application successfully created all of the geometry necessary, just like Datasmith.

Object1 C.1 contains roughly 230K entities in its STEP definition and 215 parts. Object2 C.2 has around 810K entities and 1423 parts. A side by side comparison between meshes generated by Datasmith and CADto3D from the same model can be seen in 5.1.

The values given in table 5.1 are intended to more or less show the difference between using Datasmith and CADto3D, the developed application. These values are rough approximations and give a rough order of magnitude of the operations, as these values highly depended on mesh size, tessellation quality and the used system.

For smaller files, Datasmith seems to perform well when importing compared to the developed application, especially because CADto3D requires two import processes (from the STEP file to the application and from the FBX to Unreal Engine). For larger files, Datasmith does not seem to be very optimized, and Unreal Engine stopped responding several times during import, and ultimately failed.

Regarding re-tessellation, the proposed solution of using temporary files to store shape information proved to be successful, cutting the time used in fully re-tessellating a scene in more than half, as opposed to re-importing with a different linear deflection, which is what Datasmith does.

Another advantage of CADto3D is that it creates an FBX file, which does not limit the created mesh to Unreal Engine, as it can be imported into multiple applications such as game engines and modelling software.

## 5.2 Re-tessellation

Besides re-tessellating the object completely, it is also possible to re-tessellate a single part or groups of parts individually, something that cannot be done using Datasmith.

Tessellation speed is dependent on two factors. The size and complexity of the shape directly affect the size of the temporary file used to store the shape information, and in turn how much time it takes for the tessellation to be done. Tessellation quality also affects tessellation speed, as more vertices and polygons means more time transferring the geometry from the Python process to Three.js and more time creating the scene.

The table in 5.2 presents re-tessellation times for two parts of Object1 C.1 at multiple linear deflection values. It is possible to see that tessellation speed depends widely on the model used. Even though the Front part at 0.5 linear deflection has more polygon that the Body part at 5, it didn't take as long to re-tessellate since it is a simpler and smaller surface. It is also possible to see how much geometry can be reduced using small steps of linear deflection, with 0.5 models being much faster to tessellate and producing much less geometry compared to 0.1 models, without loss of visual accuracy as seen in 5.2 and 5.3.

Table 5.2: Re-tessellation results for multiple parts and groups of parts, with number of polygons (F), vertices (V) and time used (T - average of ten samples in seconds)

| | 0.1 | | | 0.5 | | | 1 | | | 5 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | F | V | T | F | V | T | F | V | T | F | V | T |
| Front | 68K | 34K | 8.637 | 5344 | 2660 | 0.876 | 2152 | 1064 | 0.551 | 604 | 290 | 0.469 |
| Body | 364K | 182K | 110.5 | 35K | 18K | 2.925 | 17K | 8K | 1.722 | 4782 | 2343 | 1.102 |

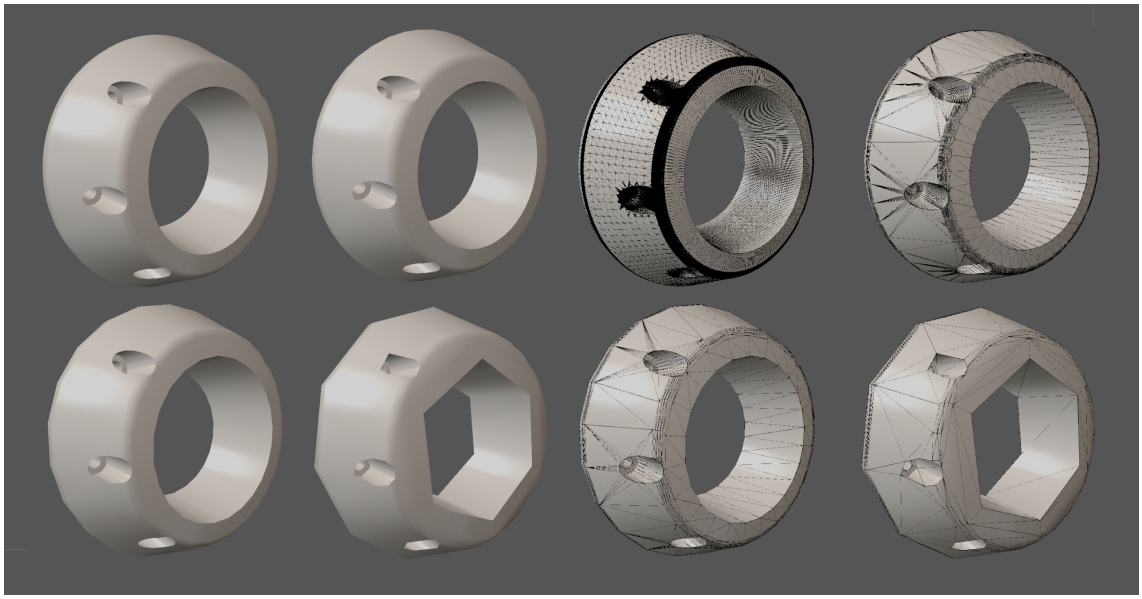

Figure 5.2: Comparison between Front parts (5.2 at different linear deflections with wireframe (from left to right, top to bottom, 0.1, 0.5, 1, 5)

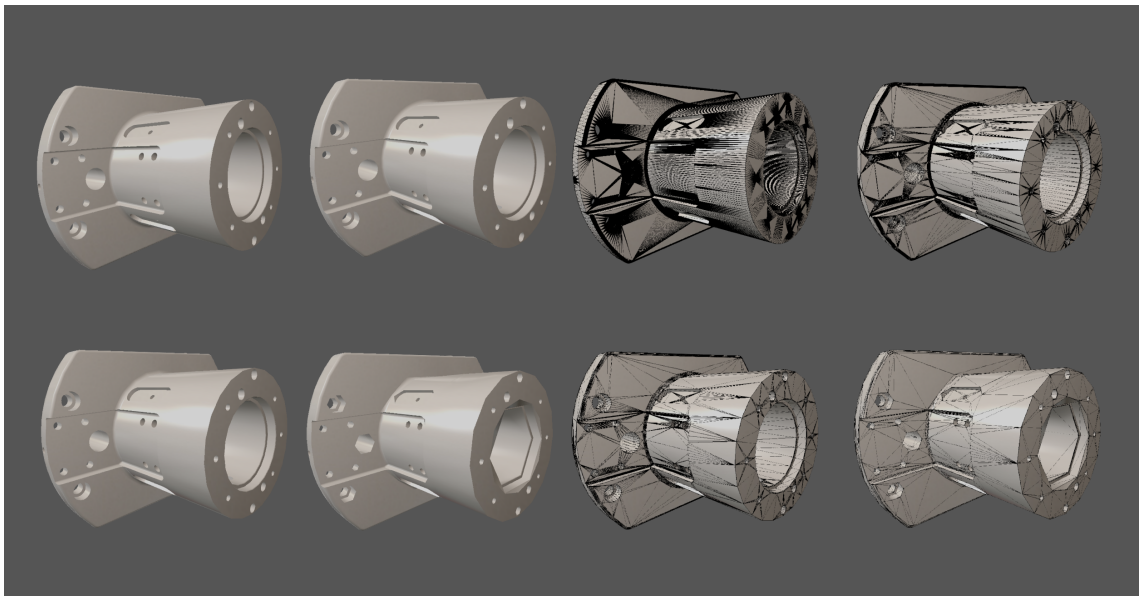

Figure 5.3: Comparison between Body parts (5.2 at different linear deflections with wireframe (from left to right, top to bottom, 0.1, 0.5, 1, 5)

## 5.3 UV calculation

Unlike tessellation, where every part of every model should be able to be tessellated, not every part can have its UVs calculated properly by the proposed algorithm in its current state, if at all.

Limitations of the algorithm include continuous closed surfaces (no normal discontinuities at all), meshes with very complex geometry, surfaces of revolution with holes and irregular surfaces of revolution.

It is often that tested parts fall into these limitations, but, at the very least, the algorithm proposed can speed up the unwrapping process of simpler parts. For the more complex parts the designer can import the FBX file into a 3D modelling software and unwrap the mesh themselves.

Examples of correct and incorrect UV mappings created by the proposed algorithm automatically and with assistance from the user can be seen in appendix C.2.

Even though the method proposed produces good results for many parts some problems were identified:

- Surface of revolution that are not straight produce UVs that have curved boundaries like seen in C.7, which causes grid textures to not line up.
- Dividing by the discontinuities of the normals often produces good patches but can also produce too few or too many patches, limiting the assistance of the user or requiring them to do too much work joining them, respectively C.11.
- Irregularities in the tessellation can lead to the method not finding the flow of the mesh C.10, producing bad seams.

- Mapping cylinders with holes in them is impossible since the algorithm cannot recognize what is a hole and what is a main boundary.

## 5.4 Evaluation

Evaluation of the application by the designers and developers of Abyssal was focused on the results obtained, the usability of the solution and if it met the requirements established in the beginning or not.

Overall, they were happy with the state of the application and the work done, especially with features that were considered important, such as the selection and re-tessellation of individual parts, parts with the same type and same material. The performance and interactivity of the re-tessellation process was also seen as a very positive aspect of the application.

On the negative side, import and loading times weren't considered to be particularly fast, even with the implemented methods to try to improved user experience (loading animation and progressive display of parts). This was especially true for very large files (>1GB), where the reading of the file by PythonOCC was regarded as taking too long, with the users not even reaching the phase where the parts start getting presented on the screen.

Regarding the UV mapping method, the results obtained were considered to be promising, but ultimately the proposed method still presented too many limitations and wasn't robust enough to be used commercially. For simpler shapes the UV maps created were deemed acceptable, but for more complex ones the UVs obtained weren't really usable, which meant that the designers would have to redo them manually.

In the end, there is some room for improvement, but the final solution met the set expectations, notably due to the amount of work done and the time-frame for the development of the project.

# Chapter 6

# Conclusions and future work

## 6.1 Goal satisfaction

Overall, the final result of this dissertation is a working solution that fulfills the main requirements set in the beginning.

The obtained application can import STEP files, tessellate models and export FBX files, while providing a user friendly interface, better re-tessellation performance than Datasmith and, in general, more options for the user to control the characteristics of the final mesh.

All of the most important functional requirements were completed. FR-16, 17, 18, 20, 21 (3.1.1) were ultimately deemed unimportant in the context of this dissertation and were not implemented. Regarding the non-functional requirements, the performance of the application is quite good when compared to Datasmith as seen in section 5.1. It is also responsive to user interaction, and provides good feedback to actions by displaying the loading animation and progressively rendering parts on import.

The proposed method to calculate UV coordinates generates valid maps, with the main contribution in this step being the preparation process that allows the application of the BFF algorithm to meshes that don't have disk topology. Both the hole-filling method and the cutting method with detection of mesh flow allow the generation of UV maps for meshes that normally could not have BFF applied to them, while still having acceptable distortion.

In the end, the objectives for this project were accomplished and an innovative solution was created that can efficiently convert CAD models into geometry in a game engine, with suitable tessellation and UV maps, and proper part hierarchy and materials.

## 6.2 Future work

The developed application is a good prototype that provides the basic features required but is far from being a complete commercial application. Common features such as renaming parts, re-organizing groups, and deleting elements are still to be implemented.

Regarding the importing and tessellating processing, the application is entirely dependent on PythonOCC, which, besides requiring a different Python version than FDK SDK, also provides no control over how tessellation is done in different parts of the mesh. Ideally, a custom STEP reader and tessellator could be developed in the future, with a focus on increasing performance, even though this is not a trivial task. Another option to explore would be the ability to import more file types than just STEP, including properietary formats.

Where there is more room for improvement in the developed application is in the automatic UV method.

Firstly, the division of the mesh in patches, even though it produces decent results, occasionally sub or over divides the mesh, which prevents the user from interacting fully with the subdivisions or requires too much work to join adjacent patches, respectively. In situations that not enough patches are created the better approach would be to divide the mesh where normals vary over a certain threshold in a groups of vertices, instead of just using single vertex discontinuities. For over patching, the solution could be to limit patch area, relative to object size, or limit the angle between adjacent patches. Another solution could be to allow the user to set where the mesh should be divided by selecting the seam edges directly.

The other problem faced is that surfaces of revolution are cut in sub-optimal ways if they are not tessellated regularly or have holes in them. To handle irregularly tessellated mesh, the solution would be to rework how the flow of the mesh is obtained and develop a method that can detect edge loops in triangular meshes, or approximations of these loops since they not always exist. For surfaces of revolution with holes, the process would be to identify what is a hole and what is an outer boundary, applying the hole-filling technique already defined, and cut the mesh accordingly. One option to identify holes could be to compute the normal of the face created by the boundary of the hole, and compare it to the tangent of the surface alongside the boundary.

Finally, since the UV mapping method proposed often requires user interaction to obtain better results, an improvement could be the automatic merge of adjacent patches. This process could either be done using analytical methods like those in Autocuts and Optcuts, or machine learning algorithms based on the data of how users merge patches in the application.

# References


[Bak05] Timothy J. Baker. Mesh generation: Art or science? *Progress in Aerospace Sciences*, 41(1):29 – 63, 2005.

[BLP+13] David Bommes, Bruno Lévy, Nico Pietroni, Enrico Puppo, Cláudio T. Silva, Marco Tarini, and Denis Zorin. Quad-mesh generation and processing: A survey. *Comput. Graph. Forum*, 32:51–76, 2013.

[BO91] M.S. Bloor and J. Owen. Cad/cam product-data exchange: the next step. *Computer-Aided Design*, 23(4):237 – 243, 1991.

[Chr] Chromium inter-process communication(ipc). https://www.chromium.org/developers/design-documents/inter-process-communication. Accessed: 2019-06-17.

[Cun06] Steve Cunningham. *Computer Graphics: Programming, Problem Solving, and Visual Communication*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[Data] Datasmith overview. https://docs.unrealengine.com/en-us/Studio/Datasmith/Overview. Accessed: 2019-06-05.

[Datb] Using datasmith with cad file formats. https://docs.unrealengine.com/en-US/Studio/Datasmith/SoftwareInteropGuides/CAD. Accessed: 2019-06-05.

[DRV98] C. Demartini, S. Rivoira, and A. Valenzano. *Product data exchange using STEP*, pages 257–269. Springer US, 1998.

[Elea] Electron application architecture. https://electronjs.org/docs/tutorial/application-architecture. Accessed: 2019-06-17.

[Eleb] Electron remote. https://electronjs.org/docs/api/remote. Accessed: 2019-06-17.

[FBX] Fbx file overview. https://www.autodesk.com/products/fbx/overview. Accessed: 2019-06-06.

[Fle01] G. Flegg. *From Geometry to Topology*. Dover Books on Mathematics. Dover Publications, 2001.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1990.

REFERENCES


[Gre14] Jason Gregory. *Game Engine Architecture, Second Edition*. A. K. Peters, Ltd., 2nd edition, 2014.

[III04] Jerry O. Talton III. A short survey of mesh simplification algorithms. 2004.

[Inc08] The Khronos Group Inc. COLLADA – Digital Asset Schema Release 1.4.1. Standard, 2008.

[ISO95] Industrial Automation Systems and Integration - Product Data Representation and Exchange - Part 21: Implementation Forms: Clear Text Encoding of the Exchange Structure. Standard, International Organization for Standardization, 1995.

[Jyl10] Jukka Jylänki. A thousand ways to pack the bin – a practical approach to two-dimensional rectangle bin packing, 2010.

[LKK+18] Minchen Li, Danny M. Kaufman, Vladimir G. Kim, Justin Solomon, and Alla Sheffer. Optcuts: Joint optimization of surface cuts and parameterization. *ACM Transactions on Graphics*, 37(6), 2018.

[LMV02] Andrea Lodi, Silvano Martello, and Daniele Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1):379 – 396, 2002.

[Lo85] S. H. Lo. A new mesh generation scheme for arbitrary planar domains. *International Journal for Numerical Methods in Engineering*, 21(8):1403–1426, 1985.

[LT97] Kok-Lim Low and Tiow-Seng Tan. Model simplification using vertex-clustering. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 75–ff., New York, NY, USA, 1997. ACM.

[MB08] Kenton McHenry and Peter Bajcsy. An Overview of 3D Data Content, File Formats and Viewers. Technical report, University of Illinois at Urbana-Champaign, 2008.

[MM11] D.A. Madsen and D.P. Madsen. *Engineering Drawing & Design*. Cengage Learning, 2011.

[OBJ] Obj file specification. https://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf. Accessed: 2019-06-06.

[Opt] Optim documentation. https://optimdocumentation.z22.web.core.windows.net/. Accessed: 2019-06-05.

[PiXa] Pixyz. https://www.pixyz-software.com/. Accessed: 2019-06-05.

[PiXb] Pixyz plugin. https://www.pixyz-software.com/plugin/. Accessed: 2019-06-05.

[PiXc] Pixyz studio. https://www.pixyz-software.com/studio/. Accessed: 2019-06-05.

[PiX18] Tata Elxsi benchmark report: comparing PiXYZ Studio and Unreal Datasmith. Benchmark report, Tata Elxsi, 2018.

[PTH+17] Roi Poranne, Marco Tarini, Sandro Huber, Daniele Panozzo, and Olga Sorkine-Hornung. Autocuts: Simultaneous distortion and cut optimization for uv mapping. *ACM Trans. Graph.*, 36(6):215:1–215:11, November 2017.

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximation for rendering complex scenes. pages 455–465, 01 1993.

[Ros97] Jarek Rossignac. Simplification and compression of 3d scenes, 1997.

[SC17] Rohan Sawhney and Keenan Crane. Boundary first flattening. *ACM Trans. Graph.*, 37(1):5:1–5:14, December 2017.

[SCOGL02] Olga Sorkine, Daniel Cohen-Or, Rony Goldenthal, and Dani Lischinski. Bounded-distortion piecewise mesh parameterization. In *Proceedings of the Conference on Visualization '02*, VIS '02, pages 355–362, Washington, DC, USA, 2002. IEEE Computer Society.

[SFH05] Michael S. Floater and Kai Hormann. *Surface Parameterization: a Tutorial and Survey*, volume 1, pages 157–186. 01 2005.

[She99] Jonathan Richard Shewchuk. Lecture notes on delaunay mesh generation. Technical report, 1999.

[She02] Jonathan Richard Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry*, 22(1):21 – 74, 2002. 16th ACM Symposium on Computational Geometry.

[SPR06] Alla Sheffer, Emil Praun, and Kenneth Rose. Mesh parameterization methods and their applications. *Found. Trends. Comput. Graph. Vis.*, 2(2):105–171, January 2006.

[SS15] Jason Smith and Scott Schaefer. Bijective parameterization with free boundaries. *ACM Trans. Graph.*, 34(4):70:1–70:9, July 2015.

[STL] Stl file specification. http://www.fabbers.com/tech/STL_Format. Accessed: 2019-06-06.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70, July 1992.

[Tan14] J. Tanton. *Encyclopedia of Mathematics*. Facts on File Science Library. Facts On File, Incorporated, 2014.

[The] Theia optim. https://theia.io/optim/. Accessed: 2019-06-05.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, s3-13(1):743–767, 1963.

# REFERENCES

# Appendix A

# Extract of STEP file

```
ISO-10303-21;
HEADER;
FILE_DESCRIPTION (( 'STEP AP214' ), '1' );
FILE_NAME ('Part3.STEP', '2019-01-11T13:05:26',
    ( 'UGJOEB-' ), ( 'UGJOEB-' ), 'SwSTEP 2.0', 'SolidWorks 2017', '' );
FILE_SCHEMA (( 'AUTOMOTIVE_DESIGN' ));
ENDSEC;

DATA;
#1 = ORIENTED_EDGE ( 'NONE', *, *, #246, .T. ) ;
#2 = AXIS2_PLACEMENT_3D ( 'NONE', #603, #464, #796 ) ;
#3 = CARTESIAN_POINT ( 'NONE',
  ( -8.660254037844387300, 5.000000000000000000, -10.000000000000000900 ) ) ;
#4 = AXIS2_PLACEMENT_3D ( 'NONE', #642, #207, #568 ) ;
#5 = EDGE_CURVE ( 'NONE', #607, #600, #711, .T. ) ;
#6 = LINE ( 'NONE', #433, #660 ) ;
#7 = EDGE_CURVE ( 'NONE', #214, #248, #210, .T. ) ;
#8 = LINE ( 'NONE', #398, #663 ) ;
#9 = FILL_AREA_STYLE ('',( #321 ) ) ;
#10 = VERTEX_POINT ( 'NONE', #720 ) ;
#11 = ORIENTED_EDGE ( 'NONE', *, *, #60, .T. ) ;
#12 = DIRECTION ( 'NONE',
  ( 0.000000000000000000, -0.000000000000000000, 1.000000000000000000 ) ) ;
#13 = EDGE_LOOP ( 'NONE', ( #612, #397, #267, #540 ) ) ;
#14 = AXIS2_PLACEMENT_3D ( 'NONE', #208, #715, #337 ) ;
#15 = VECTOR ( 'NONE', #491, 1000.000000000000000 ) ;
#16 = AXIS2_PLACEMENT_3D ( 'NONE', #402, #43, #156 ) ;
...
#821 = COLOUR_RGB ( '',
  0.000000000000000000, 1.000000000000000000, 0.000000000000000000 ) ;
ENDSEC;
END-ISO-10303-21;
```

Extract of STEP file

# Appendix B

# Example of a halfedge mesh structure

```
{
    boundaries: {
        Face {halfedge: Halfedge, index: 0}, ...
    },
    corners: {
        Corner {halfedge: Halfedge, index: 0}, ...
    },
    edges: {
        Edge {halfedge: Halfedge, index: 0}, ...
    },
    faces: {
        Face: {halfedge: Halfedge, index: 0}, ...
    },
    halfedges: {
        Halfedge: {
            corner: Corner,
            edge: Edge,
            face: Face,
            index: 0,
            next: Halfedge,
            onBoundary: false,
            prev: Halfedge,
            twin: Halfedge,
            vertex: Vertex
        }, ...
    },
    vertices: {
        Vertex: {halfedge: Halfedge, index: 0}, ...
    }
}
```

Example of a halfedge mesh structure

# Appendix C

# Multiple results

## C.1 Objects created by CADto3D in Unreal Engine

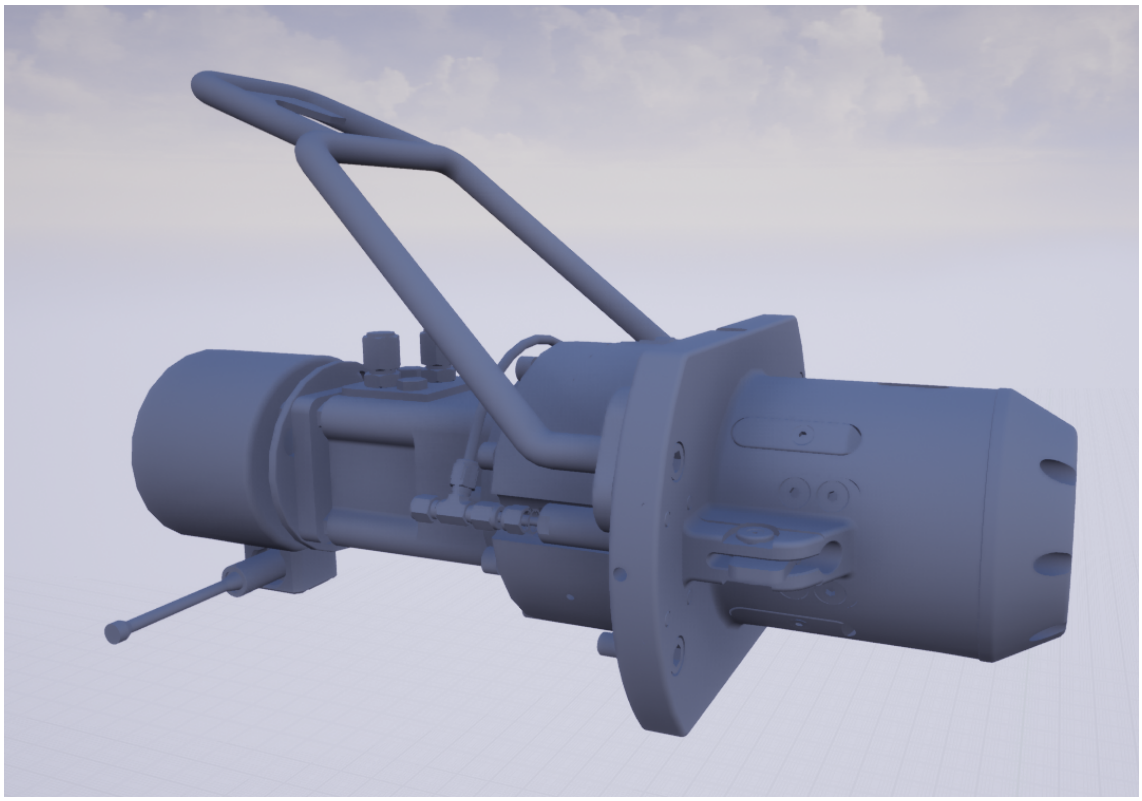

Figure C.1: Object1 of 5.1 in Unreal Engine

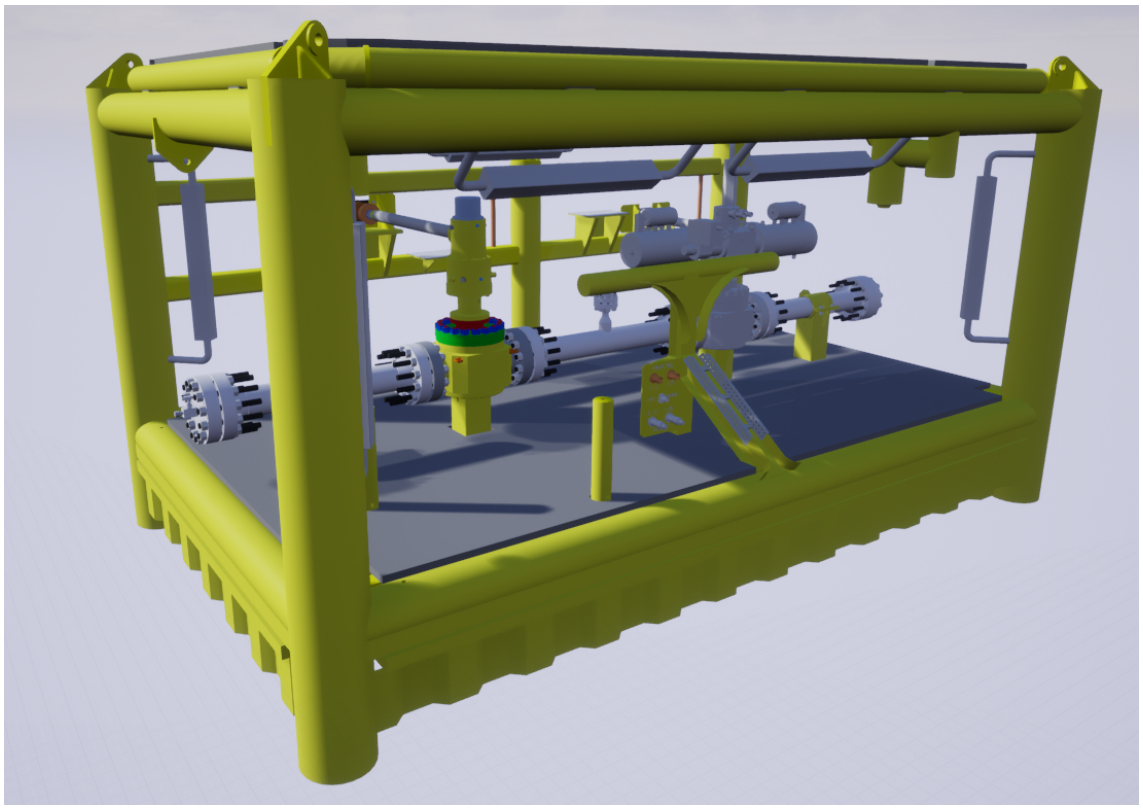

Figure C.2: Object2 of 5.1 in Unreal Engine

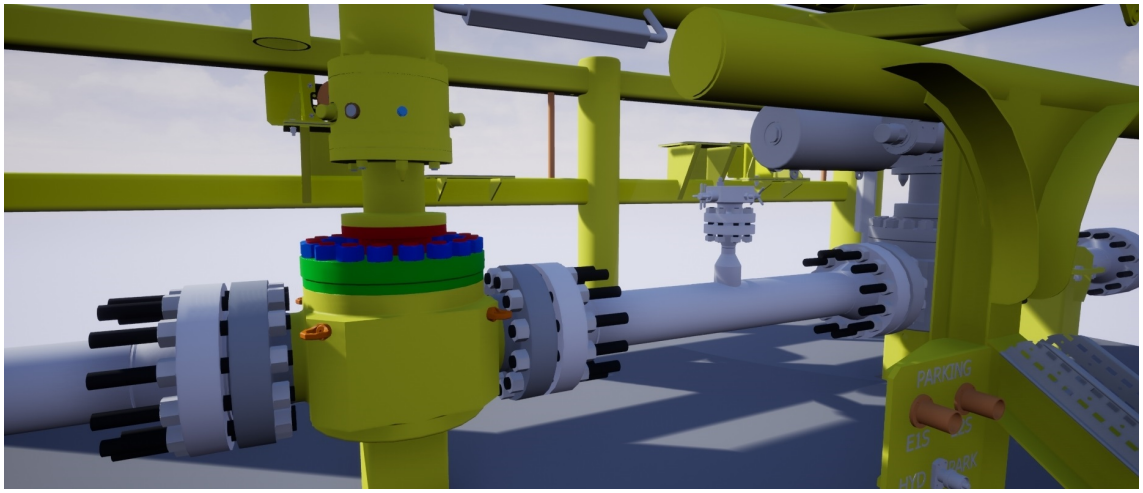


Figure C.3: Detail of Object2 of 5.1 in Unreal Engine

## C.2 Calculated UVs

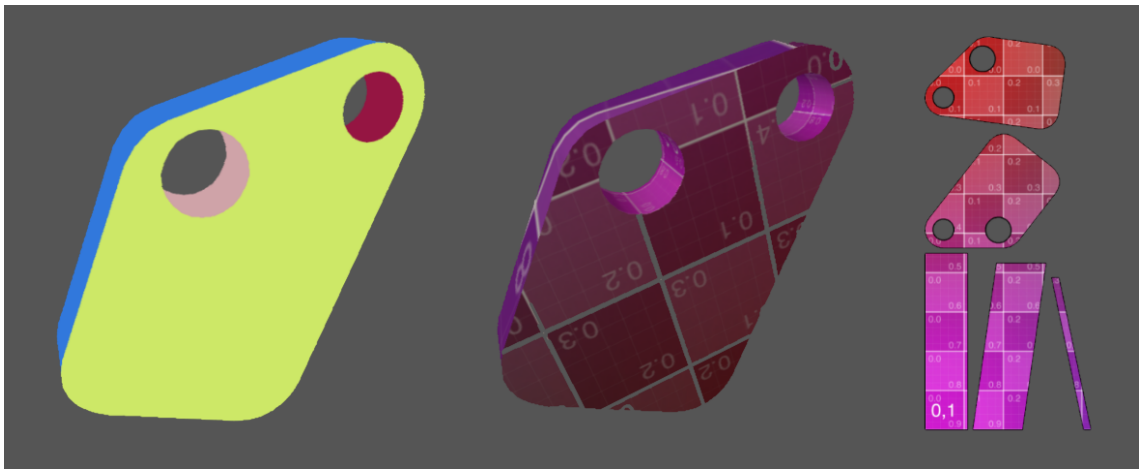

Figure C.4: Patching, UVs and texture atlas of a part automatically unwrapped by the algorithm

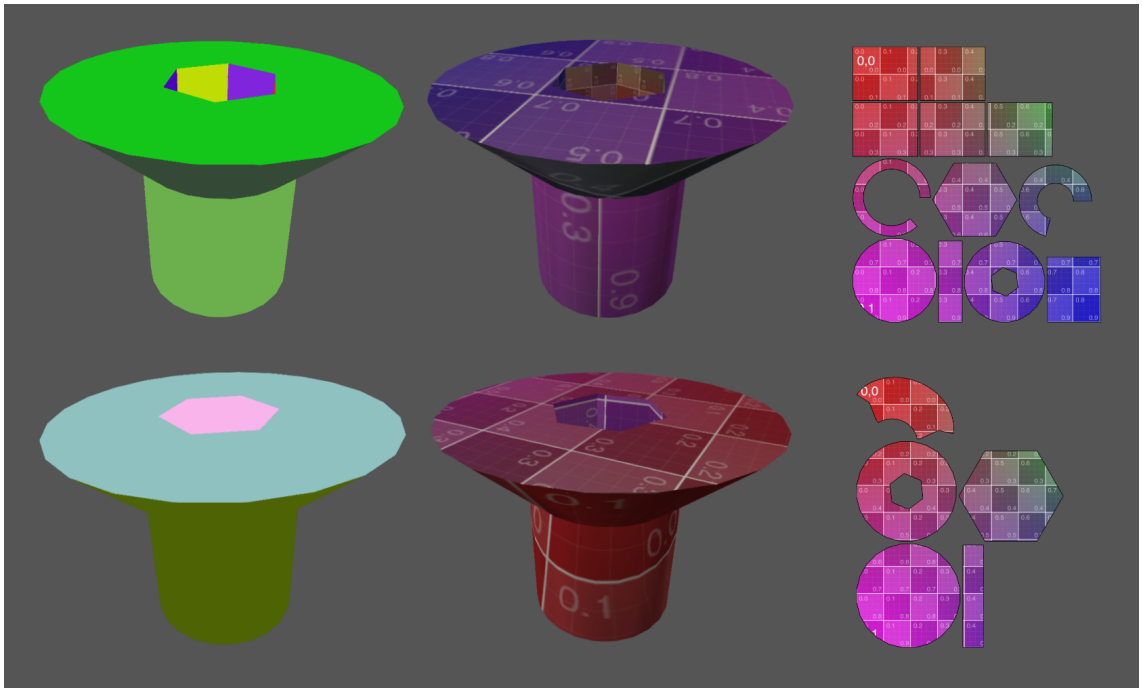

Figure C.5: Patching, UVs and texture atlas of a screw without user assistance (top) and with (bottom). Notice the difference between hole-filling (holed circle) and cutting (cylinder).

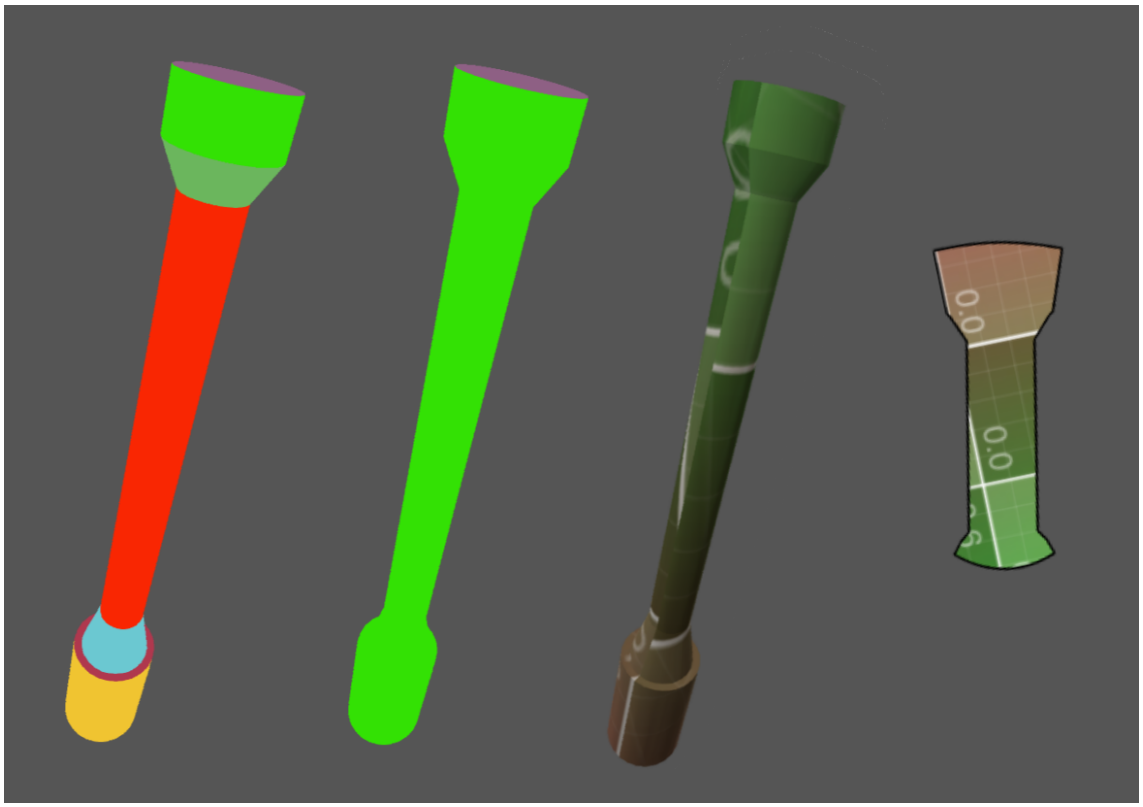

Figure C.6: The method used can detect the flow of the mesh even when joining patches.

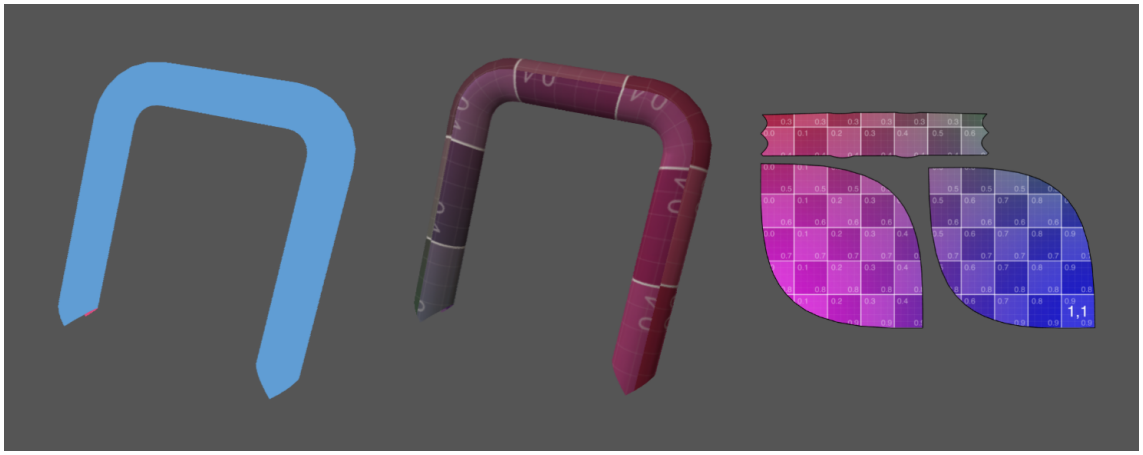

Figure C.7: The developed method can create seams that follow the flow of the mesh even when the surface is curved.

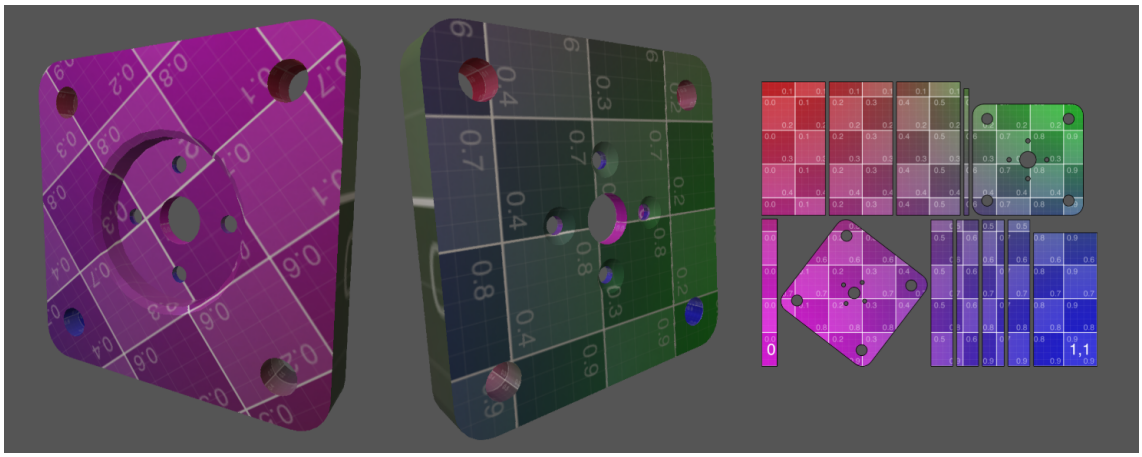

Figure C.8: Multiple holes in the same patch are also dealt with correctly.

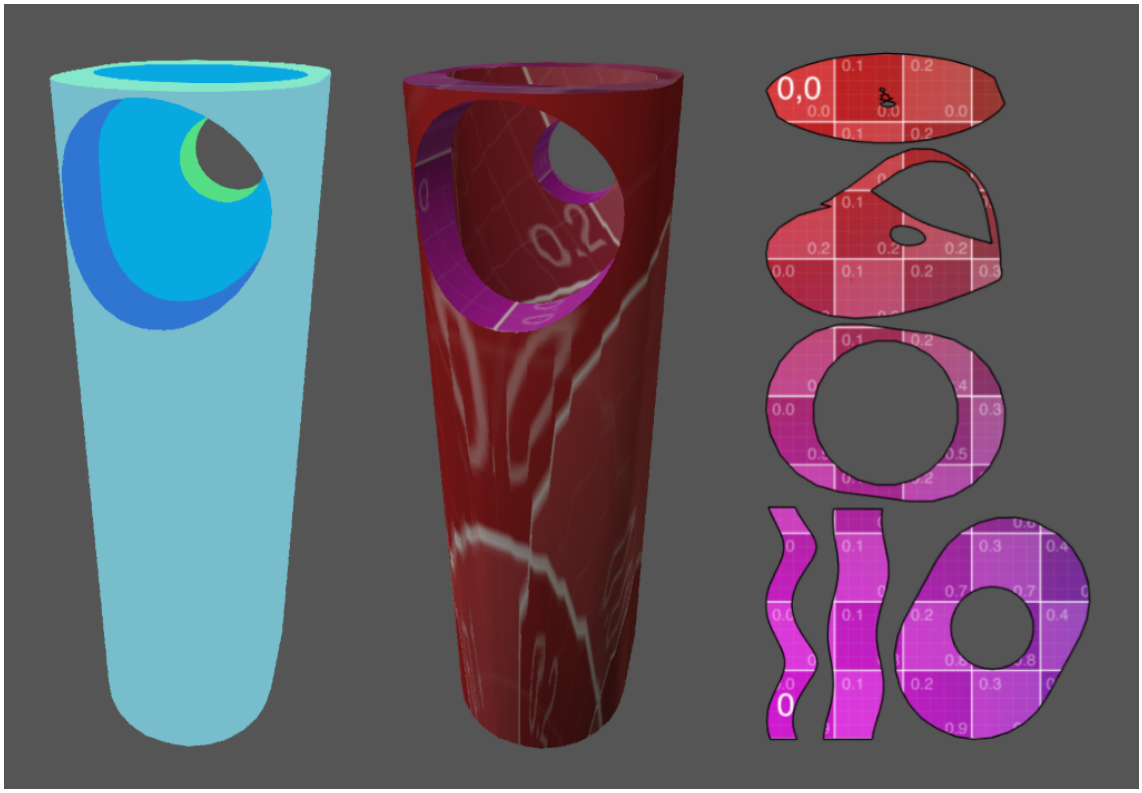


Figure C.9: For cylinders with holes the results aren't the best as all boundaries except for the main one is filled

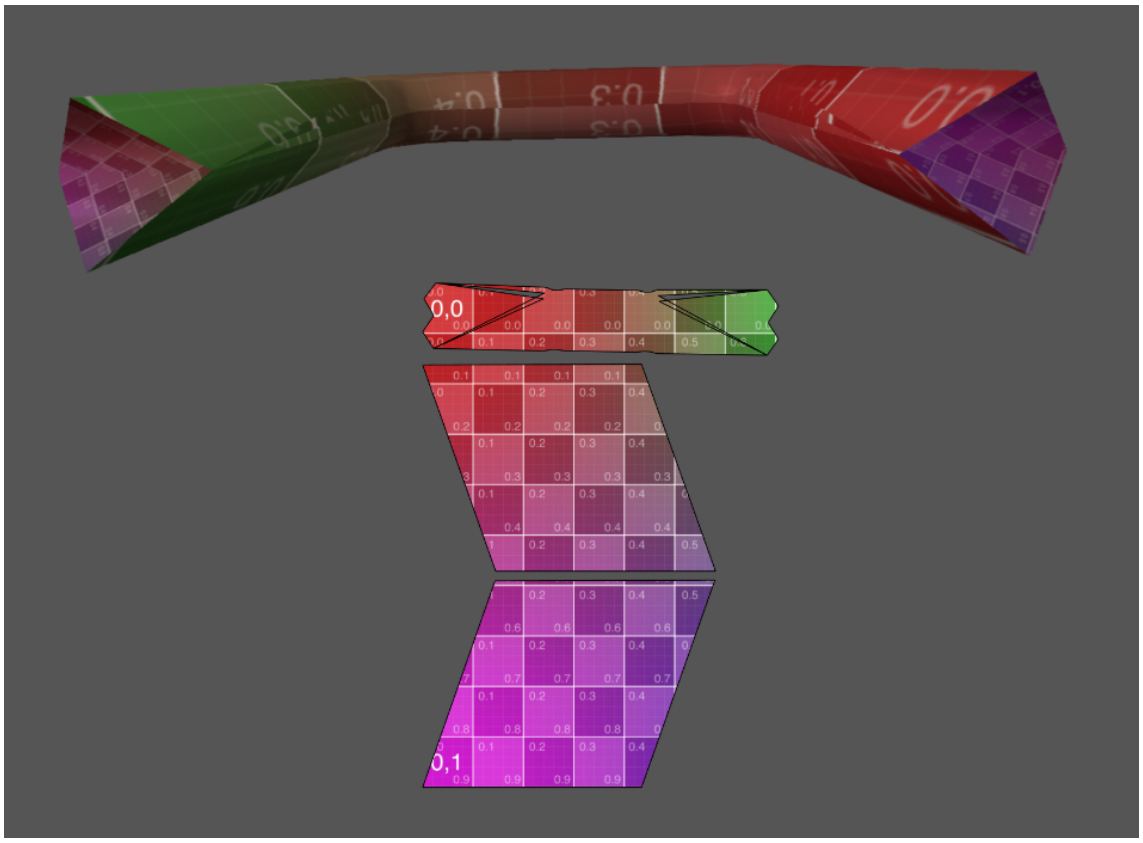

Figure C.10: Artifact created in the same cylinder as C.7 with a higher value of linear deflection. Surfaces with few polygons can become irregular leading to sub-optimal UV mappings.

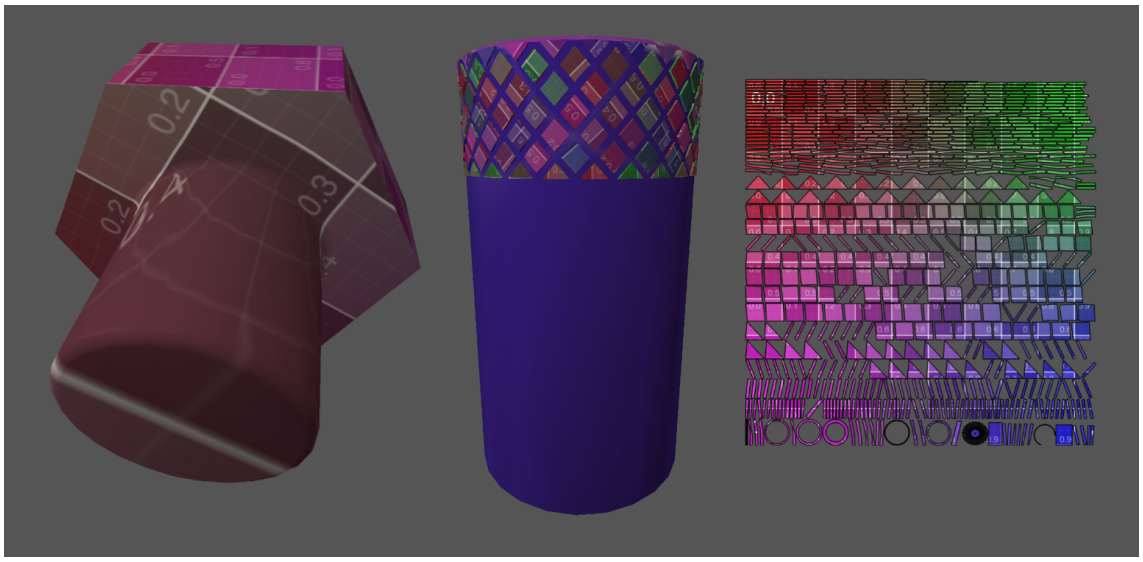

Figure C.11: The division of the mesh by normals can lead to two opposing problems: the sub and over creation of patches (left and right respectively).

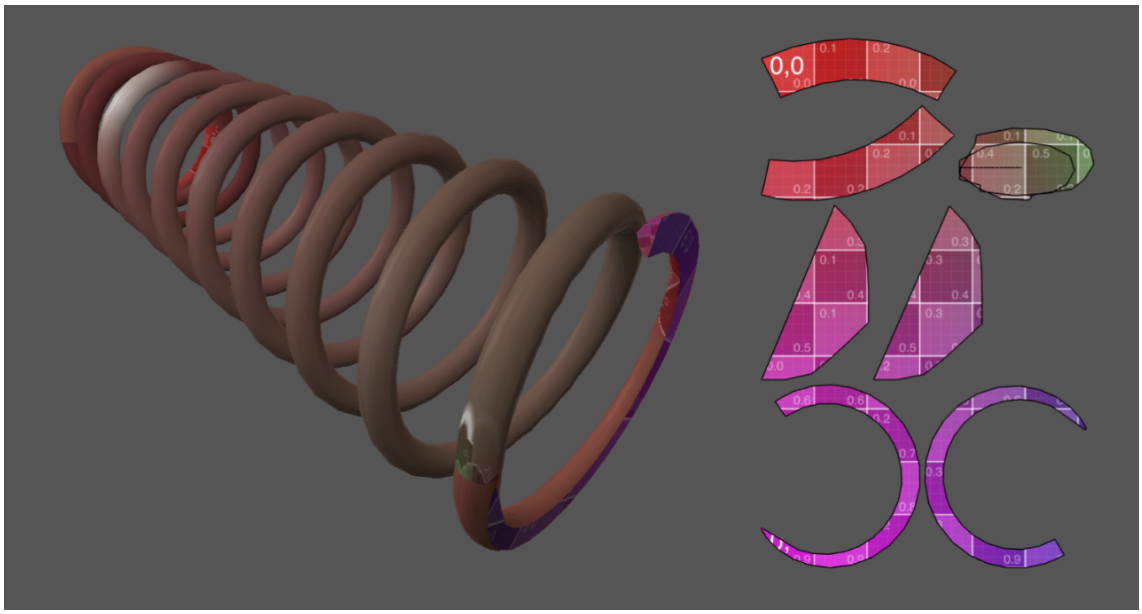


Figure C.12: When complex meshes are able to produce results they are often not the best.