School of Computing
FACULTY OF ENGINEERING
AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Low-poly Mesh Generation for Any Model Input

Thomas Jackson

**Submitted in accordance with the requirements for the degree of
MSc High Performance Graphics and Games Engineering**

2023/2024

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Source Code (Deliverable 1 & 2)	GitHub Repository: (Appendix A)	Supervisor, Assessor, (12/08/2024)
Project Report (Deliverable 3)	Report	Supervisor, Assessor, (12/08/2024)

Type of project: Exploratory Software

(Signature of Student) Thomas Jackson

Summary

The computation of low resolution meshes for level of detail applications in the games industry is an important part of the optimisation pipeline. The inclusion of meshes exceeding tens of thousands of triangles is becoming increasingly common within the video game industry. For this reason, the computation of low resolution meshes for level of detail application is now needed more than ever. This project builds an application that can be used for this purpose, taking any input mesh and converting it into a low-poly version.

This report explores the subject area of low-poly mesh generation, highlighting methods used and the variety of approaches taken to solve this problem. The research highlights that an isosurface extraction approach followed by a mesh simplification process will produce the desired results. Using these pre-existing techniques, requirements are produced alongside a high level design overview outlining the produced application. Chapter 4 details the implementation of the software, explaining each step necessary to build each application. Further details on the methods used to debug and validate the implementation are also presented in this chapter. To determine the success of the project, the implementation is evaluated against the key project aims producing tabulated and visual results. The results presented show that the project is a success, presenting visually similar low-poly meshes when compared to their original inputs. Comparing the outputs to those produced by industrial tools shows that the application produces similar results though in a much slower time.

The report ends with the conclusion stating that the aims of the project have been met with a valid low-poly mesh generating application produced. Further work is proposed to optimise the application so that it can compete with industrial tools. Aside from this, the adaptation of the signed distance field program to tolerate self-intersecting triangles is identified for future improvements. Details on the application usage and source code are available in Appendix A or directly at the GitHub repository: `https://github.com/Tomizzed2001/LowPolyMeshGeneration`.

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Hamish Carr, for his guidance throughout the project with both planning and technical advice. Secondly I would like to thank my assessor Dr. Markus Billeter for his mid-project advice on how I could approach the final stages of my project.

Contents

Chapter 1

Introduction

The rapid advancement in the visual quality of games over the last decade can be partially attributed to the inclusion of higher definition models made of tens of thousands of triangles. Visualisation of these models is a computationally expensive task due to the large number of polygons needed to represent such high levels of detail. This makes the optimisation of such rendering techniques paramount to the games industry to ensure acceptable performance in gameplay. One such technique simplifies the models that are not within direct view of the player, lowering their render cost without drastically altering the graphics of the gameplay. Unfortunately, real-time methods cannot always produce a significant reduction in the number of triangles. Instead, lower-poly models have to be pre-computed and swapped at runtime to decrease rendering cost. This report will detail the production of an application to produce such low-poly models and repair any topological issues with a given input mesh so that it can be rendered at minimal cost. This opening chapter will detail the aims and motivation of this project, alongside the objectives and deliverables that should be met. It will further comment on any ethical, legal, or social issues that may arise through the completion of this project.

1.1 Project Aim

The aim of this project is to create a software application for converting highly detailed models with high triangle counts into similar looking meshes with a much lower polygon count. The produced application will aim to generate manifold output files with significantly fewer triangles within a reasonable time frame when given a large mesh.

A manifold mesh is defined as being a fully closed, self-intersection free mesh that has a single cycle around each vertex [6]. Manifold meshes are often desired for the games industry, particularly for object and character models. This is because non-manifold geometry can cause visual artifacts when rendered due to holes and triangle folds. Additionally, non-manifold geometry can make rendering and mesh operations unnecessarily expensive due to the incomplete/incorrect data structure. However, some meshes used in the games industry are non-manifold on purpose, for example, a terrain is often left open at the base as it is out of view of the renderer and has clearly defined boundaries. Closing such a mesh would only increase performance costs and complicate the program and the same can be applied to items like foliage. This program aims to be used where manifold geometry is preferred alongside a lower polygon count rather than terrains or foliage.

Reasoning the core aims of this project, as the demand for quality graphics rises in video games, artists are forced to use more triangles to create higher levels of detail. Despite this creating a visually enjoyable experience for players, higher polygon counts increase render cost which reduces the smoothness of gameplay. This means that developers of games have to consider the visual quality to performance trade-off so that games can be released without an excessive demand for top of the line hardware. To assist in this issue, techniques that reduce the number of polygons whilst retaining visual quality are highly sought after in the industry. Methods such as tessellation shaders, aim to reduce the level of detail (LOD) in a mesh at runtime based on the distance of a mesh from the camera. This utilises the idea that objects not in direct view of the player require as much detail and therefore do not require as much computation. This project aims to build on this concept and reduce polygon counts to a much lower level so that they can be integrated with the optimisation pipeline. Whilst it would be ideal that this method could also be used in real-time, it is computationally expensive to ensure the mesh is manifold and reduce the polygon count to the lowest level. Therefore, the only time requirement of this project is that the entire process be completed within a reasonable time frame considering the size of the input mesh. The reason for this ambiguous statement is that large meshes will take considerably more time than smaller ones. Furthermore, differences in hardware will result in time differences making it difficult to give an exact time for all computations to be done.

1.2 Objectives

This section will list the objectives of this project with respect to the project aims.

1. Explore the subject area of low-poly mesh generation to gain insight into possible solutions and techniques that may be applied for this project.
2. Produce a detailed design overview for the application which can be used during implementation.
3. Implement the low-poly mesh generating program.
4. Evaluate the implemented software on a set of 10 meshes, determining the number of polygons reduced, execution time, visual similarity and resulting topology.
5. Compare and contrast outputs of the software against applications used in the industry.

1.3 Deliverables

This section will list the deliverables to be produced throughout the course of this project.

1. An application that produces an output mesh with significantly reduced polygon count compared to the input.

2. A GitHub repository containing the source code of the application and instructions for use (Appendix A).

3. A report documenting the research, design, implementation and evaluation processes undertaken throughout the project.

1.4 Ethical, legal and social issues

Whilst this project has no applicable ethical or social issues associated with the work, legal issues could arise given the subject area. These issues regard the usage of copyrighted model data for testing which could be acquired online. To avoid this, only open-source or freely available data with the correct licensing information will be used in this project. Furthermore, all data used for this project from external sources will be cited with reference to the original creators.

Chapter 2

Background Research

This chapter of the report will explore the topic of low-poly mesh generation, reviewing academic literature on the subject to build insight into previous work done at a fundamental level and the current state of the art. The information presented in this chapter will then be used alongside the aims of this project to attain a set of methods and techniques which will be used to produce the results required.

To explore the surrounding literature, material of relevance has been found through graphics conference proceedings such as SIGGRAPH [12], pre-existing literature surveys on mesh simplification [21, 19, 27], and search engines like Google Scholar. By sourcing research from different locations it is hoped coverage of the topic will be varied and balanced. Furthermore, work that has been accepted into high-ranking conferences and journals is likely to be of high quality. Despite this, research papers will be critiqued for validity and usefulness with regard to the project's aims.

2.1 Literature Review

Mesh simplification is a well explored topic of computer graphics with a broad variety of methods explored and evaluated over the years. This section of the background research will focus on 4 categories of low-poly mesh creation, evaluating both early and recent work. Challenges posed by each method will be presented and evaluated in the context of this project's aims.

2.1.1 Direct Mesh Simplification through edge collapse

Many of the techniques used to reduce the number of polygons within a mesh are applied directly to the input mesh. Early work in the area, such as the edge collapse method by Hoppe et al. [15] focused on collapsing edges in the pre-existing mesh in an order specified by error cost. Garland and Heckbert [13], and Hussain et al. [16] built upon this method, employing different error cost calculations and ways of finding the optimal point to collapse to. Garland and Heckbert define the quadric error metrics (QEM) method which associates a collapse with an error based on the sum of squared distances to each plane of a vertex. They then calculate the optimal vertex position using a 4x4 quadric error matrix and use pair contraction to collapse the edge. Conversely, Hussain et al. [16] choose to collapse one vertex of the edge into the other, avoiding the need to compute and store optimal vertex positions.

Each paper [15, 13, 16], is successful in reducing the polygon count of the input mesh, though each has its issues when considering the objectives of this project. The original edge collapse method outlined by Hoppe et al. [15] is often criticised for being slow due to how it calculated the error cost. Considering that the work was done between 1993 and 1996, this algorithm could struggle to produce timely results when confronted with modern-day models that may exceed a million triangles. Furthermore, despite papers [13, 16] presenting fast and visually preserving outcomes, they fail to guarantee a manifold result and are reliant on the input mesh being manifold. Whilst this may be a complete solution when dealing with models that are not required to be manifold, this project is aimed at models that would benefit from being manifold. It is for this reason that these are incomplete solutions for this project, although the techniques applied may be suitable for simplification once a manifold mesh is acquired.

2.1.2 Visual-based mesh generation

Recent work by Gao et al. [12] presents a method that utilises visual hulls to generate extremely low poly models for buildings. Utilising the visual hull technique allows non-manifold inputs to be converted and still produce a manifold result. Furthermore, by using different viewpoints of a camera, the output remains similar despite the drastic decrease in polygons. Whilst this method works well for building models, applying it to any model can result in undesired outcomes. This highlights that for more complex models, different techniques would need to be used and therefore it is unlikely that the tools outlined in this paper should be used for implementing this project. However, the paper details extensive evaluation steps specific to the low-poly meshing industry. They include the comparison between their model and other software on a consistent dataset of models. This highlights that it will also be important for this project to be compared alongside software hinting at steps for the evaluation.

2.1.3 Machine learning methods

Whilst this report will not pursue any of the methods utilising machine learning techniques, much of the current work in mesh simplification uses them. The reasoning for this choice is due to a lack of background knowledge in the subject of artificial intelligence and the large number of resources needed for training such a model. An example of the work in this area is the paper by NVIDIA [14] which utilises a rendered representation of the mesh to find matches to simplified lookalikes. This produces successful results, particularly on models that contain foliage which is often difficult to simplify due to its non-manifold and dense structure. Another recent paper that uses machine learning is the paper by Potamisa et al. [33] which samples points of the mesh into sets. These sets are then fed into an edge prediction model to build a mesh with a lower triangle count. They acknowledge that whilst they are successful in simplifying the mesh, a manifold topology cannot be guaranteed.

2.1.4 Isosurface Simplification

When attempting to guarantee a manifold surface, it is often the case that an isosurface is constructed from the original mesh. Whilst isosurfaces are not low-poly by default, due to them often being highly tessellated, they are simplified at the end of the application to reduce complexity. The paper by Zoë et al. [40] focuses on exactly this. It generates an isosurface with the Marching Cubes algorithm (by Lorensen and Cline [24]) and then simplifies the topology by removing the excess handles and reducing the genus. Consequently, this action also reduces the polygon count in many cases, though not to a degree that it could be called low-poly. Whilst this is not directly applicable to the application this project strives to create, the application of the marching cubes algorithm to produce a manifold surface is a concept that should be explored.

In similar and more recent work, Manson and Shaefer [25] present a method of isosurface extraction that results in a lower polygon count whilst preserving features that are often lost on an isosurface. During the standard marching cubes algorithm [24], sharp features are often lost since they cannot be represented within a cube. To avoid this, and reduce the number of triangles, a dual vertex method is used in which vertices are paired and placed directly on the isosurface causing some triangles to become redundant. This results in an accurate representation of the original mesh, due to the preservation of features, which also has a much lower polygon count. The solution presented by Manson and Shaefer may meet the success criteria for this project. Furthermore, the application of further simplification algorithms, like the ones outlined in Subsection 2.1.1, could be applied to gain an even lower polygon count. Following a similar thought process to this, the Robust Low-Poly Meshing paper by Chen et al. [8] presents a solution that generates extremely low poly, feature preserving, manifold meshes. Chen et al. acknowledge that whilst Manson and Shaefer present a good solution, the number of triangles can be far too great to consider low-poly. In their method, they instead present a 2 step process consisting of isosurface extraction and mesh optimisation. Within the first step they employ unsigned distance fields partnered with a modified version of the MC33 algorithm [9] to extract an isosurface. They then apply a variation of techniques to regain the features lost in the extraction process. The optimisation stage includes a set of simplification techniques including the quadric edge collapse method [13] to bring the solution to the desired number of triangles. The advantage of the more modern method by Chen et al. in 2023 over Manson and Shaefer in 2010 is that they obtain the same benefits in addition to an extremely low poly mesh that does not require further simplification.

All 3 methods [8, 25, 40] present satisfactory results when regarding this project's aims. Furthermore, the results of the paper by Chen et al. show it as equal or better to many of the tools used to generate low-poly meshes used in the industry such as Simplygon [37] and Blender [5]. With this paper being recently published in 2023, given the time of writing, this can be considered to be state of the art. Whilst the results are promising, the complexity of the paper may mean that this method cannot be followed exactly within the given time frame. Despite this, the general outline of the isosurface extraction followed by simplification seems to produce the results needed. Furthermore, less recent methods using isosurfaces also generate valid results suggesting that they will be a useful tool for this project.

2.2 Methods and Techniques

Considering the project aims in Section 1.1 and the background research done, valuable techniques for this project will include isosurface extraction and mesh simplification techniques as used in [8]. Furthermore, it will be important to consider the different data structures required to implement the techniques efficiently. This section will cover each of these methods and the full extent of techniques required to implement them.

2.2.1 Isosurface extraction

Extracting an isosurface involves the Marching Cubes algorithm first presented in the 1987 paper by Lorensen and Cline [24]. Using this method guarantees a manifold, self-intersection free mesh when given a signed distance field that details the distance from uniform points on a grid to the original mesh. By incorporating this step into the implementation, even non-manifold input meshes can be returned manifold -as is one of the requirements of this project. This subsection will detail the different methods of obtaining the signed distance field and the variations of the marching cubes algorithm.

Distance Fields

There are two types of distance fields that can be computed, signed and unsigned. Unsigned distance fields are simply the distance from a point p to the closest point on the mesh [19]. The signed distance field is then the distance as defined previously with the addition of a positive or negative sign. This sign is determined where a distance is negative if the value is inside the mesh and positive if outside the mesh [2].

To calculate the closest distance from a point to a triangle, Jones [18] presents a 3D and a 2D method and compares the efficiency of the two. Both methods involve dividing the triangle into regions as demonstrated in Fig. 1 of [19] and determining which region the point lies. The 3D method explained, projects the point in space onto the plane of the triangle. This is achieved by taking the angle between the triangle normal and the vector describing a triangle vertex to the point in space. Contrasting this, the 2D method defines a transformation matrix which translates and rotates the triangle to lie on the yz plane. This makes the x component of the point's location irrelevant when determining the region it lies in. Jones concludes that the 2D method completes faster than the 3D method in many scenarios. However, it should be noted that the 2D solution requires the storage of a 4x4 transformation matrix and several 3D vectors defining edge normals for each triangle in the mesh. Furthermore, test data shows that the largest mesh tested contained only 3080 triangles. This is a reasonably small mesh when considering that some modern meshes exceed 1 million triangles, with most having more than 10,000. Eberly [10] presents an alternative method for calculating the distance from a point to a triangle. This technique also splits the triangle into regions in which the point may lay. However, these regions differ from the one shown in [19], with the regions at each edge being defined by the lines of the neighbour edges. Eberly presents a more mathematical method by defining the distance in terms of a quadratic function with the aim of the algorithm being to minimise it. For functions that have their minimum on the boundary, the regions are used to find the distance to the closest point. Whilst Eberly presents a clearly defined implementation, the work is not clearly evaluated so no conclusions can be made from the results of the work. Despite this, the paper presents another option for defining the regions of the triangle which does not require extra computation to find the normals for each edge.

The survey on distance fields by Jones et al. [19] explains that there are multiple ways to calculate the sign of the distance from a triangle. The simplest of these methods is to use the dot product of the face normal and the vector from triangle to point, defining negative values as being inside the mesh. However, as detailed by Payne [32], distances of opposite signs can be tied, particularly at sharp areas of the mesh. This is due to the normals of the vertices and edges being defined the same as the face, which is not necessarily true in most cases. To fix this, normals must be assigned for each vertex and edge separately; these are called pseudonormals or in some cases smooth normals. Baerentzen et al. [3] use an angle weighted pseudonormal whereby each face incident on a vertex or edge has its normal summed to produce a normal vector. This produces a smoothed normal vector allowing points to be accurately classified as inside or outside the mesh during tied distance scenarios. It is also explained that for edges, an unweighted average of both faces produces correct pseudonormals.

Further research into distance fields highlights that there are limitations to the technique, particularly when dealing with self-intersecting triangles. Chapter 34 of GPU Gems 3 [28] explains that triangle folds can cause an issue with signed distance fields.

Whilst it does not exactly explain how this can be fixed due to it being a complex task requiring multiple passes of the distance fields, it details the criteria for which it becomes a problem. It describes triangle folds as overlapping triangles with backward facing normals. This causes issues with signed distance fields as it prevents the outside and inside of the meshes from being clearly defined. Similarly, self-intersecting triangles can cause the same issues as they prevent there from being a clearly defined outside and inside of the mesh.

Marching Cubes

In 1987, Lorensen and Cline suggested the marching cubes algorithm [24] to create triangulated models that could be used to visualise medical data. The general algorithm used the concept that a signed distance field could be defined as a set of cubes with a signed distance for each of the 8 vertices of the cube. Using this data, triangle vertices could be placed on each edge where the signed distance was negative on one side and positive on the other. This is because the sign flip implies that the edge is intersected by the original surface as one point is inside the mesh and the other outside. They then explain that the number of possible cases can be limited to 2^8 or 256 cases. By applying rotations and symmetries to the cube, they surmised that there could only be 14 different base cases within the 256 and thus, the binary representation of the cube could be used to select a case from a look-up table and triangulate the cube. The final step is then to march through the signed distance field applying this method until the entire field is defined and a resultant mesh is found, hence the name "Marching Cubes". They further improve this model by applying linear interpolation along each edge where a vertex is placed. This positions each vertex according to where the surface intersects, resulting in a smoothed surface as opposed to one clearly defined by cubes.

Soon after the publication of the marching cubes method, researchers discovered that the 14 cases proposed by Lorensen and Cline could result in a non-manifold mesh with holes in it. Montani et al. [26] states that base cases 3, 6, 7, 10, 12, and 13 have ambiguous faces. They define a face on a cube as ambiguous if diagonally opposite vertices of the face have opposite signs. This results in the complimentary cases of each configuration listed to fit incorrectly with the other configurations. To fix this, Montani et al. present a modified look-up table with differing triangulations for the compliments of the ambiguous cases. This results in a guaranteed manifold surface at the cost of the maximum number of triangles per cube increasing from 4 in [24] to 5 in [26]. Several years later, Nielson [29] proposed that the number of cases could be further extended to represent the multitude of possible surface representations within each cube. This technique used more than the original 256 configurations used in [24, 26] and added additional lookup tables for each ambiguous case. To decide on which specific sub-configuration should be used, Nielson employs the use of his earlier work, the asymptotic decider [30] and determines how vertices are connected on the interior of the cube. Nielson then reasons that this combination of cases is extensive with no other cases being possible. This is then further backed up by the alternative proof presented by Carr [7] in 2007, completing the extensive research into possible case configurations for marching cubes.

Further research into the area of marching cubes focused on feature preservation with the marching cubes algorithm. The first of these solutions was by Kobbelt et al. [23] where they proposed the extended marching cubes algorithm to retain the sharp features lost in the marching cubes process. By using a directed distance field, they detect which of the cubes in the grid would contain a sharp feature and apply additional sample points to the grid to re-construct the original shape. Unfortunately, the method presented can produce gaps in the mesh, with the authors acknowledging this in their concluding statement. Another notable process is the dual marching cubes method by Schaefer and Warren [35]. This method also preserves sharp features with the addition of producing a water-tight mesh. To implement this, they use octrees to define the mesh and extract a dual grid from this. The usage of a dual grid then allows them to re-create the sharp features lost during the marching cubes algorithm thus creating a hole free, feature preserving mesh.

2.2.2 Mesh Simplification

Mesh simplification is the main thing to consider when trying to generate a low-poly mesh from a high-poly mesh. As discussed in Subsection 2.1.1, many of the earliest methods of low poly mesh generation were only mesh simplification. With this, the standard procedure was to calculate an error cost for an edge collapse and then do the least expensive operation as seen in [13, 16, 15]. This subsection will briefly detail the different options to simplify meshes and re-iterate the techniques in subsection 2.1.1.

Edge collapse methods

The paper by Hoppe [15] outlines the original edge collapse method in which a vertex is collapsed into another via an edge joining them both. The remaining edge is then moved to an optimal position, removing the edge and redefining the faces around it. It is highlighted that choosing the edge to collapse correctly is crucial to maintaining the quality of the simplified meshes. To choose this order, they propose a distance energy term where the squared distance to the mesh is calculated for the change in each edge. Hussain [16] argues that this method is slow due to the multiple distance to mesh calculations per edge and proposes the half edge collapse method. The half edge collapse method similarly collapses an edge, however, it chooses one of the original vertex positions to remain the same. This means vertices do not need to be moved and instead an edge can be costed without calculating optimal vertex positions. One of the most popular methods for edge collapse and mesh simplification is in the paper by Garland and Heckbert [13]. This paper uses quadric error metrics to calculate the error of an edge collapse and an optimal vertex so that the collapse method defined by Hoppe [15] can be used. (This method is covered in detail in subsection 2.1.1 for more details on the implementation). This method is presented as both fast and relatively memory conserving, only needing 10 floating point values per vertex. Furthermore, this method results in visually similar meshes as shown in the figures presented in papers [13, 8].

Vertex removal methods

Another method of mesh simplification is to ignore the edges and focus on the removal of vertices. This technique often results in non-manifold meshes requiring that once enough vertices are removed, the mesh is re-triangulated. This method is called vertex clustering and is presented by Rossignac and Borrel [34]. Vertex clustering is the process by which a mesh is separated into multiple bounding boxes which are then generalised to a single vertex. Each of the generalised vertices are then joined together through re-triangulation to create an approximation of the overall mesh with significantly fewer triangles. The disadvantage to this method is that the resulting mesh can be non-manifold and result in "shark fin" like triangles.

Shroeder et al. [36] implement a vertex decimation technique whereby all vertices in the mesh are evaluated against certain criteria and removed if the criteria are met. These checks vary depending on how the vertex is classified as shown in Figure 1 of [36], with certain vertices using a distance to plane criteria whilst others use distance to edge. If eligible for collapse, the vertex and all triangles using the vertex are removed from the mesh. The resultant hole is then re-triangulated using a recursive procedure which aims to reduce the number of triangles used previously whilst maintaining the topology of the mesh. A disadvantage to this method is the recursive reconstruction of each hole which can be time consuming and does not guarantee a set number of polygons will be removed.

2.2.3 Data structures

Each of the well-known data structures used in computer graphics is outlined in Chapter 2 of the book by Botsch et al. [6]. The first set of data structures discussed is the face based data structure, the first of which is triangle soup. Triangle soup is an unordered set of triangles stored in an array represented by the vertices defining the 3 points of a triangle. This method of storage is the most basic though it does not represent the topology of the mesh and consumes excess memory with vertex locations being defined multiple times. To solve this issue, the indexed face data structure is proposed with an index given to represent each vertex. Each face is then defined as the set of indices referring to each vertex on the face. Many of the indexed face file formats are used in industry with the most notable being the OBJ file format. This format is also capable of storing vertex normals and texture coordinates though is not required for the format to be used.

Aside from face based structures, edge based ones can be used, particularly when wanting to know about the connectivity of a mesh. One of the data structures specified in [6] is the winged-edge structure by Baumgart [4]. This method sees that each edge stores the vertices it is between, the faces it joins and the next/previous edge in the array. This is particularly useful when doing edge operations as both affected faces are known. Equally, this can cause issues when trying to find all connected vertices to a single vertex. Two similar methods proposed are the half edge [20] and the directed edge [39] data structures. Each of these splits an edge into 2 halves, one for each triangle, with the distinction being that the directed edge structure focuses on the direction of an edge. Faces are defined by their edges in both structures, with the half edge structure keeping track of the next and previous edges visited. The directed edge structure avoids doing this by storing each edge in the order of the faces making it indexable with modulo 3.

2.3 Choice of methods

Choosing the set of techniques to be implemented requires the consideration of the aims of the project and the time frame in which it is to be done. Starting with the first method of generating a distance field in Subsection 2.2.1. The paper by Jones [18] is chosen as a guideline for the implementation. Utilising Jones' method for dividing a triangle into regions over the method presented by Eberly [10], provides a more detailed estimate of which area of the triangle the point is closest to. This is because using Eberly's method will require further computation if in an edge region as a point to line segment calculation will be required rather than the standard point to line one. Furthermore, returning from the loop earlier in the case the distance is found is always better, meaning that the broadening of the vertex regions makes it more likely that this option will be chosen, saving on computation. In addition to this, the 3D method for calculating the distance when inside the triangle will be used for its simplicity, as it only requires a single dot product and a multiplication. For deciding which region the point lies in, a novel algorithm will be used consisting primarily of cross products and dot products to determine which side of a plane a point lies. By doing this it is hoped that the excess computation of the 3D method by Jones can be avoided and the added memory cost of the 2D method can be reduced. The final step to produce the signed distance fields will be to choose the sign of the distance. Given the problems with using face normals to calculate the sign, the angle weighted pseudonormal method by Baerentzen et al. [3] will be used. Choosing this method will guarantee that points are classified as inside or outside the mesh correctly for meshes where self-intersections are not present. In the cases where self-intersections / triangle folds are present, the solution will not be able to guarantee a correct signed distance field. Whilst having the functionality to deal with this would be ideal, this is a much more complex task, as outlined in Chapter 34 of [28], and would take too much time to implement.

For the second stage of the iso surface extraction, a marching cube method must be chosen. For this project the case tables presented by Montani et al. [26] will be used alongside the original marching cubes method by Lorensen and Cline [24]. Choosing these tables will mean that the iso-surface is guaranteed to be manifold unlike the ones presented by Lorensen and Cline. Furthermore, Nielson's tables [29] result in more than 256 cases making the choice of triangulation complex. Utilising the tables by Montani et al. ensures that the result will be manifold, as is required of the project, whilst only using 256 cases keeping it simple to compute. Reasoning the choice of Lorensen and Cline's marching method, it is by far the simplest approach and given the time frame available, a working solution is prioritised over an advanced partial solution. Whilst the feature preservation qualities of methods [23, 35] are nice and will result in better looking meshes, it is only necessary that meshes look similar from a distance where sharp features are less noticeable.

Moving on to the choice of simplification techniques, the method presented by Garland and Heckbert [13] will be implemented. This method was chosen for its speed advantage over the original edge collapse method by Hoppe [15] and for the simplicity of the error cost over the one proposed by Hussain [16]. Furthermore, with the input to the collapse process being guaranteed to be manifold, processes that take the borders of the mesh into account are unnecessary for this application. It is worth noting that the vertex removal methods [36, 34] were removed from consideration due to the need to re-triangulate the surface post vertex removal. This is a costly operation which can be entirely avoided with the use of edge collapse techniques without impact (in most cases) on visual quality or computation time.

Knowing which methods will be used, the best data structures can be chosen. This project will make use of the index faced data structure, namely the OBJ file format, which is one of the most popular within the industry [1]. This format will primarily be used for input and export of the meshes making the program easy to use without prior data formatting. For operating on the mesh, once it has been converted to an isosurface, the directed edge format will be used. The benefit of using this structure is that it contains more detail on the connectivity than the face based formats. This is done whilst maintaining the detail of both edge and vertex connectivity without the storage of excessive data. This will lower memory usage and the computation required to update the structure per collapse operation.

To summarise, the distance to triangle method presented by Jones [18] will be used alongside a novel selection method for the region of the triangle to find the distance to such a triangle. To calculate the sign of this distance, an unweighted pseudonormal will be calculated for each edge and vertex in the mesh as shown by Baerentzen et al. [3]. To build the isosurface from the signed distance field, look-up tables by Montani et al [26] will be used in conjunction with the marching method by Lorensen and Cline [24]. Finally, the mesh will be simplified with the use of the quadric error metric technique discovered by Garland and Heckbert [13]. This combination of methods and techniques will result in a manifold, low-poly mesh being computed from any given input mesh within a reasonable time thus, meeting the success criteria for this project.

Chapter 3

Software Requirements & System Design

Building on the chosen technologies for this project, selected in Section 2.3, this chapter will outline a set of requirements and explain design decisions made prior to implementation. The first section will detail the functional and non-functional requirements of the implementation, expanding on the aims of the project detailed in Section 1.1. The design section will then build upon the requirements and explain the high level design decisions made, detailing how the user will operate the application.

3.1 Software Requirements

The requirements of the project aim to encapsulate the overall aim (Section 1.1) and provide specific metrics to create detailed success criteria for the produced application. In addition to this, requirements are made taking into account the background research (Chapter 2) to ensure that all requirements are attainable. Despite this project utilising a renderer to visualise the output meshes, the goals of the project are specific to the mesh generation. Considering this, requirements are based solely on the meshing process as it is expected that users will view the output meshes in their preferred environment for their specific use cases.

3.1.1 Functional Requirements

This subsection will focus on detailing the functional requirements of the implemented software. The program should be mostly autonomous with user input only being required to input the desired input files and operating parameters. This means that the user will not be involved for most of the process meaning functional requirements will detail what each application takes as input and then produces. Ensuring that each of the functional requirements are met, will mean that the program produced will output results, though they will not necessarily be successful.

As explained in the system design (Subsection 3.2.1), the application will be split into 3 sub-programs, each focusing on the different processes identified by the research (Section 2.3). For this reason, functional requirements are identified with 2 numbers (e.g, 2.3) within Tables 3.1, 3.2, 3.3. The first number signifies the process with 1 being distance field generation, 2 being the marching cubes process and 3 being the simplification step. The second number is then simply the index of that requirement within the process. For example, 2.3 would refer to the 2nd requirement of the marching cubes process.

FR Index	Requirement Description
1.1	The program should take a .OBJ file as input and read in the associated mesh.
1.2	The program shall allow users to optionally specify a maximum grid size for the largest dimension, x y or z.
1.3	The program will generate a grid of uniform cubes which encapsulates the entire mesh.
1.4	The program will keep the user informed of the current stage of the process to indicate the ongoing computation.
1.5	The program will output a signed distance field to a human-readable format.

Table 3.1: The functional requirements (FR) table for the signed distance fields application.

FR Index	Requirement Description
2.1	The program should take a signed distance field, such as the one output by the distance field application as input.
2.2	The program should allow users to optionally specify if an OBJ file output is required alongside the directed edge format.
2.3	The program should store no more than one vertex per edge of the distance field.
2.4	The program will output interpolated vertex positions for each edge of the grid that the surface crosses.
2.5	The program will output an isosurface to the directed edge file format.

Table 3.2: The functional requirements (FR) table for the marching cubes application.

FR Index	Requirement Description
3.1	The program should take in an isosurface in the directed edge file format.
3.2	The program shall allow users to optionally specify a desired triangle count and or a maximum error tolerance.
3.3	The program will keep the user informed of the current stage of the process to indicate the ongoing computation.
3.4	The program should terminate when the user-specified triangle count is met or the maximum error tolerance has been met.
3.5	The program should return a watertight, self-intersection free mesh.

Table 3.3: The functional requirements (FR) table for the edge collapse application.

3.1.2 Non-Functional Requirements

The non-functional requirements of this program focus on the constraints in which the functional requirements should be met. Unlike the functional requirements, non-functional requirements will be specified for the whole system rather than the single application. This is because they apply to all applications or the outcome of the entire system. Furthermore, this set of requirements will focus on the quality of the solution which will be the primary focus during the evaluation of the program. Whilst meeting the functional requirements in subsection 3.1.1 will mean an outcome is produced, meeting all non-functional requirements will mean a successful outcome with regard to the project's aims outlined in section 1.1.

NFR Index	Requirement Description
1	The system should take no longer than 1 minute per 2500 polygons in the input mesh to complete.
2	The system should be modular meaning that the applications can be used multiple times without re-computing the prior stages.
3	The system should be well documented, allowing the source code to be improved upon by other developers.
4	The system should ensure the isosurface data structure maintains its topology throughout the process.
5	The system should reduce polygon counts to at least 10% of the original count when given an input exceeding 10000 triangles.
6	The system should be straightforward to use with only the compile and run instructions being necessary to operate the program.
7	The system should output a similar looking mesh to the one fed into the application.

Table 3.4: The non-functional requirements (NFR) table.

3.2 System Design

This section detailing the system design will discuss the various design decisions made to accommodate all of the requirements in section 3.1. The overall design will follow a similar method to the one detailed by Chen et al. [8] and explored in subsection 2.1.4. This design section will explain how the 2-step method detailed in [8] will be re-structured to match the aims and constraints of this project. To achieve this, the first subsection will show a full system overview and explain the reasoning behind the 3 separate applications. The following subsections will focus on the high-level design of each application, referring to the technologies chosen in section 2.3 as the building blocks for each application.

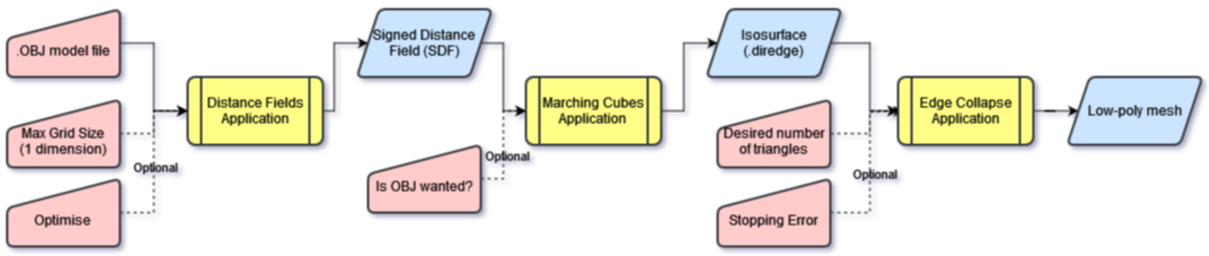

.OBJ model file
Max Grid Size (1 dimension)
Optimise
Optional
Distance Fields Application
Signed Distance Field (SDF)
Is OBJ wanted?
Optional
Marching Cubes Application
Isosurface (.diredge)
Desired number of triangles
Stopping Error
Optional
Edge Collapse Application
Low-poly mesh

Figure 3.1: Full system design. Manual inputs are pink, programs are yellow and input/output are blue.

3.2.1 Full system design overview

The full software application consists of 3 separate programs that are to be used in series to generate a low-poly model. The order in which the programs should be used is shown in Figure 3.1 and is based on the 2 core steps: isosurface extraction and mesh simplification as used by Chen et al. [8]. There are 2 key reasons for splitting the application into 3 separate programs which are detailed in the following paragraphs.

The first reason for separating the applications is that by developing each stage individually, each process can be debugged without the interference of other processes. This means that problems encountered during the development stages are limited to one application, therefore, saving time searching through the code base for bugs. Furthermore, separating each of these concerns means that application debugging can be done faster as prior stages are not computed. Another benefit to this approach, when considering debugging, is that it is much easier to feed test data into separate programs. For example, to get an output from the marching cubes algorithm, an object file would need to be fed into the distance field application to produce a scalar field. By separating the two processes, with each having its own input values, hand made distance fields can be fed directly into the marching cubes algorithm which have known and calculable outcomes. This can help when assessing the functionality of the application for the smallest cases as well as any foreseen edge cases.

The second reason for using 3 applications instead of one is to allow the user to interact with the system in a modular fashion (NFR 2). This will be massively beneficial to the user when they require multiple models with varying numbers of polygons. By separating the programs the user may re-input the isosurface with a different number of desired triangles to get a different output. This way, the user does not need to wait for the distance field and iso-surface to be re-computed every time they need a different level of detail. It is also possible for this benefit to be an inconvenience to a user who simply wishes to set the program going and come back to a complete solution. However, the ability to re-run certain stages of the process for varying outputs is something that users are much more likely to prefer. It is also possible that should a user wish to have

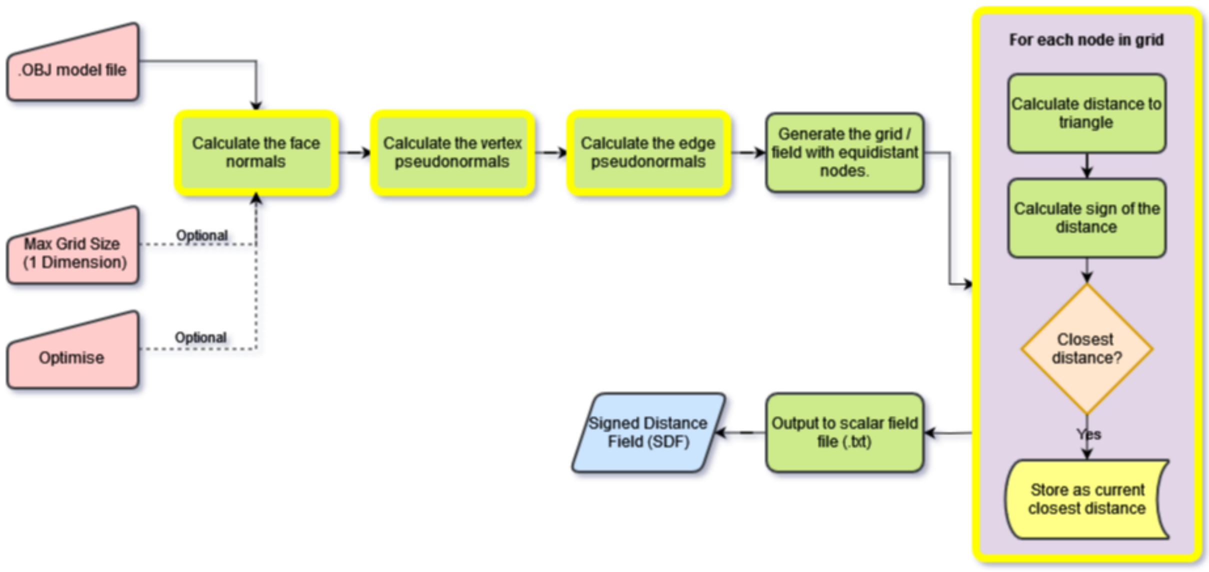

.OBJ model file
Calculate the face normals
Calculate the vertex pseudonormals
Calculate the edge pseudonormals
Generate the grid / field with equidistant nodes.
For each node in grid
Calculate distance to triangle
Calculate sign of the distance
Closest distance?
Yes
Store as current closest distance
Max Grid Size (1 Dimension)
Optional
Optimise
Optional
Signed Distance Field (SDF)
Output to scalar field file (.txt)

Figure 3.2: Block design diagram for the signed distance fields application. Parallelisable blocks are outlined in yellow.

one entire program, each of the code bases can easily be joined together to create a single application. This ties into NFR 3, which thanks to the separation, the code can be further developed or repurposed to use a single stage or multiple as they are independent of each other.

3.2.2 Distance field application design

The first stage of the process is to generate a signed distance field that can be fed into the marching cubes application to produce an isosurface [24]. As explained in section 2.3, the distance field application will use Jones's method [18] to compute the distance and Baerentzen et al.[3] to compute the sign. Considering these technologies alongside the functional requirements in Table 3.1, leads to the construction of the block diagram in Figure 3.2. As shown by the input and output blocks, the user can specify the object file (FR 1.1) and maximum grid size (FR 1.2) to obtain a signed distance field output (FR 1.5). Functional requirements 1.3 and 1.4 are more implementation dependent. However, FR 1.4 can be implicitly incorporated into the design by simply outputting to the user each time a new process is started. Further input by the user can enable optimisation of the algorithm by exploiting sparse regions of the mesh. This is further explained in the implementation of the program in Subsection 4.2.4.

Considering the non-functional requirements (NFR) within the scope of the distance field application design, the main NFR to consider is 1. Due to the application having the potential to process very large inputs and the expensive iterative computation, it can be foreseen that this application will be slow. Therefore, the design highlights processes that can be parallelised to reduce the execution time of the program.

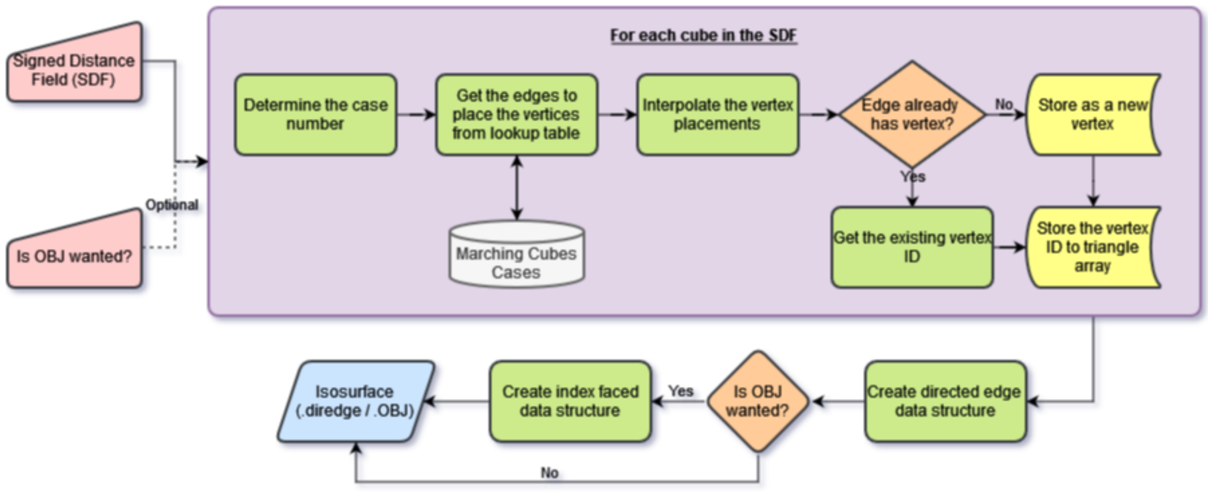

For each cube in the SDF
Signed Distance Field (SDF)
Is OBJ wanted?
Optional
Determine the case number
Get the edges to place the vertices from lookup table
Marching Cubes Cases
Interpolate the vertex placements
Edge already has vertex?
No
Store as a new vertex
Yes
Get the existing vertex ID
Store the vertex ID to triangle array
Create directed edge data structure
Is OBJ wanted?
Yes
Create index faced data structure
Isosurface (.diredge / .OBJ)
No

Figure 3.3: Block design diagram for the marching cubes application.

Processes that may be parallelised are highlighted in the diagram (Figure 3.2) with a yellow outline. Each of these components are independent from instances of themselves and should be parallelised accordingly. Whilst this does not guarantee that NFR 1 will be met, it ensures that it is at the forefront of the design where possible.

3.2.3 Marching cubes application design

Taking the signed distance field input (FR 2.1), the marching cubes application should output an isosurface to a directed edge file (FR 2.5). Furthermore, should the user choose to have an OBJ file as well, this should be outputted alongside the directed edge file (FR 2.2). Both of these functional requirements are incorporated into the design as shown in Figure 3.3. This allows the user to use the isosurface on its own; which is another benefit to the modularity of the design. The functional requirements, 2.3 and 2.4, are also accounted for within the design. Specific processes are there to ensure that only one vertex is placed per edge (FR 2.3) and that the vertex placement is interpolated between the two points as described by Lorensen and Cline [24] (FR 2.4).

The cases used for the marching cubes algorithm, chosen to be based on Montani et al [26], should be stored as an external set of data that can be referred to as a look-up table. By designing the application in such a way, cases can be easily modified, should issues with the case tables arise, as they are separated from the main code. Furthermore, this allows each cube to be quickly triangulated based on the calculated case number which is a step towards meeting NFR 1. Considering the nonfunctional requirement 4, this application creates the isosurface so it must be topologically correct. By ensuring functional requirements 2.3 and 2.4 are implemented correctly NFR 4 should follow for this stage.

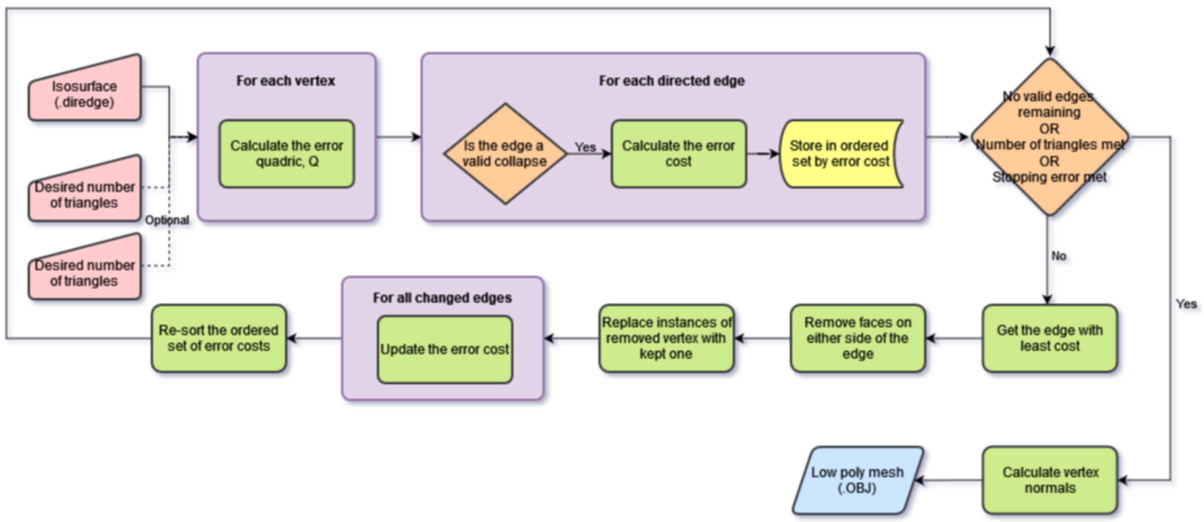

Isosurface (.diredge)
Desired number of triangles
Desired number of triangles
Optional
For each vertex
Calculate the error quadric, Q
For each directed edge
Is the edge a valid collapse
Yes
Calculate the error cost
Store in ordered set by error cost
No valid edges remaining OR Number of triangles met OR Stopping error met
No
Yes
Get the edge with least cost
Remove faces on either side of the edge
Replace instances of removed vertex with kept one
For all changed edges
Update the error cost
Re-sort the ordered set of error costs
Calculate vertex normals
Low poly mesh (.OBJ)

Figure 3.4: Block design diagram for the edge collapse application.

3.2.4 Edge collapse application design

The final step in the 3 stage process is to simplify the isosurface generated by the marching cubes program. The edge collapse application takes the isosurface as a directed edge file for input (FR 3.1) along with the optional input of a user-specified triangle count or stopping error (FR 3.2). The optional input will allow the user to tailor the output to their specific needs should they only want a moderate reduction in polygons. However, inputting a low triangle count does not guarantee that the output will have that number since the program will stop should there be no more valid edges to collapse. Figure 3.4 shows each of the manual input options alongside the processes that need to be implemented to produce a low-poly mesh (FR 3.5). The design highlights that the main loop of the application (as shown by the cyclic process) should stop should there be no valid edges, the triangle count has been achieved, or a maximum stopping error has been reached. This stopping criteria for triangle count is in place so that FR 3.4 can be met. Furthermore, ensuring that only valid edges can be collapsed ensures that the topology remains intact (NFR 4) and therefore remains watertight and self-intersection free FR 3.5. It is worth noting that this assumes the isosurface taken as input is topologically correct to begin with. The final functional requirement for this application is 3.3. Similarly to the distance fields function, larger models may result in lengthier program runtimes. To keep the user informed of progress, each process defined in the design will be a stage that can be output to the user.

Chapter 4

Software Implementation

This chapter of the report covers the technical details and methods used to fulfil the requirements detailed in the previous chapter. Starting with an insight into the data structures required to operate on and output a triangular mesh, this chapter will detail the necessary steps needed for iso-surface extraction and the simplification of the given iso-surface to produce an output mesh with a reduced polygon count. The implementation makes use of the standard $C++$ libraries such as vector and cmath as well as the well-known GLM header library for vector and matrix mathematics [11].

4.1 Data Structures

When dealing with the manipulation and storage of geometric data, it is important to consider the format that the data will take during the process. Considering the research undertaken in Subsection 2.2.3, the directed-edge data structure has been chosen for the internal manipulation of data within the application. This section will also detail the index faced data structure which is used for input and output due to its widely used format.

4.1.1 Index Faced Structure

Index faced storage of geometric meshes is one of the most common formats used by industry [1] due to how easily it can be used in rendering. An example of an index faced structure that is widely used is the Wavefront .OBJ file format. OBJ files can be exported from nearly all mesh modelling software and is accepted by nearly all modern game engines making it a stable choice for the input and output format of the application. This being said, its format does not easily allow edge operations, which are crucial to this project, so cannot be used as the only data structure.

The index faced structure stores each unique vertex in a $C++$ vector of *glm::vec3* storing the position in space of each vertex. Incidentally, each vertex's index is its position in the array and will be known as **v**. Each vertex can then be optionally given a normal (**vn**) and or texture coordinate **vt** which all have matching indexes in their respective arrays. To join these values together, each face is then defined at minimum as a set of 3 vertices making the faces array another set of *glm::vec3*, this time with length **f**, the number of triangles in the mesh.

Whilst the above paragraph completely details the structure's memory layout, the details of how this is presented in an external storage file such as the .OBJ format is slightly different. Each set of vertex data is set out in the order it is presented in the array, with **v0**, **vn0**, and **vt0** all being the first of its type written to file. They are all then connected by the face definition, where the structure of a single vertex in a triangle is defined as ***v/vt/vn***. In the scope of this application, texture coordinates will not be included in the output to simplify the overall process. An example of the input/output layout, disregarding the vertex texture coordinates, is presented in Listing 1.

```
struct indexFaced {
    std::vector<glm::vec3> vertices;
    std::vector<glm::vec3> faces;
    std::vector<glm::vec3> vNormals;
};
```

Listing 1: Code snippet of the index faced structure within the implementation.

4.1.2 Directed-Edge Structure

The directed edge data structure is a method of defining a mesh based on the direction of each half-edge in a triangle [6]. The research done in 2.2.3 shows that by using this data layout, each triangle of the mesh can be defined by 3 vertex indices. This doubles as the edge list for that triangle defined by the sending or receiving vertex. Additionally, each directed edge stores its other half which is the directed edge in the opposing direction of the complete edge.

To utilise the directed edge data structure, the spatial location of each vertex must be known and added to an array of *glm::vec3* where the x, y and z locations are stored accordingly. Thus, the size of the final vertex array will be an array of length **v**, where **v** is the number of unique vertices. Therefore, when converting directly to/from the index faced structure, the vertex array can be re-used as they are identical. Considering the vertex IDs (**vID**), the edges array, which doubles as the faces array, stores a triangle as the set of 3 vertices in counter-clockwise order around the face. Storing each set of 3 **vIDs** in a 1-dimensional $C++$ vector results in an array of size **3f** where **f** is the number of faces/triangles in the mesh. Given this array, the global edge index **eID** is simply the position in the array whilst its index within its triangle can be found as **eID % 3**;. To retrieve the face index **fID** from the array, one can do the integer division operation **eID / 3**, resulting in the floating point value being truncated and the index of the first vertex in the triangle being returned.

In addition to storing the vertices and faces, the directed edge structure differs from other methods by storing the other half of each edge in a 1-dimensional array of length **3f**. The other half of a directed edge is the directed edge in the opposing direction that would complete the edge between two adjacent triangles. Knowing this, finding the other half of an edge is as simple as searching through the faces array looking for where the same 2 vertex IDs occur but in reverse. It is worth noting that an edge can be defined by either its sending vertex or its receiving vertex, this implementation defines an edge by its sending vertex. Therefore, if **eID** is the first vertex in the triangle, then **eID** is the sending vertex and **eID + 1** is the receiving vertex for edge **eID**. Finally, whilst the first directed edge is stored as part of the data structure defined by [39], this implementation will not utilise it and therefore not store this array to save **4v** bytes of memory.

By utilising this data storage technique, each vertex is defined using 12 bytes of memory and each directed edge uses 8 bytes of memory. The cost of each face can be totalled to nothing due to it being defined by the ordering of each directed edge. This structure allows fast access to triangles and the components they are comprised of. It also makes finding each adjacent triangle easy by using the other half array. Maintaining this structure throughout the various mesh operations ensures that data can be accessed efficiently and topological correctness can be maintained as required by NFR 4. The code implementing the directed edge structure is presented in Listing 2.

```
struct dirEdge {
    std::vector<glm::vec3> vertices;
    std::vector<int> edges;
    std::vector<int> otherhalves;
};
```

Listing 2: Code snippet of the directed edge structure within the implementation.

4.2 Signed Distance Fields

The first step in constructing an iso-surface is to construct a signed distance field that partitions a 3-dimensional space into a grid of points. Within this grid, the distance field acts as a function which takes a position as input and outputs the closest distance to the mesh as a scalar value. Effectively, this means finding the closest triangle to a point for every point within the grid. The implementation of the signed distance fields detailed in this section follows the design outlined in Section 3.2.2, taking an .OBJ file as input to produce a signed distance field in the form of a 3-dimensional set of floats.

4.2.1 Choosing a grid

The size of the grid is determined by two key factors: the size of the mesh in 3D space and the spacing between grid points. To calculate the size of the grid required, the maximum and minimum x, y and z positions are stored whilst the input file is read into the program. This ensures that the full extent of the mesh, as positioned on all 3 axes, is encapsulated by the field as per FR 1.3. Note that the x, y and z values are compared independently so the smallest x and largest x will be stored regardless of their y and z values. As a precaution, padding of 2 grid lengths is added to each value so that the mesh does not touch the walls of the scalar field. Once the bounds of the grid have been determined, the field is partitioned into an integer number of grid points using the size of the grid calculated or defined by the user. It may be necessary to round the x, y and z values so that the spacing is completely even between all points. An example of the code required to position the start and end points for even spacing is shown in Listing 3.

```
for(int i = 0; i < 3; i++){
    if(point[i] < 0){         // Ensure the co-ordinate stays negative
        point[i] = -(abs(point[i]) + (sizeOfGrid - fmod(abs(point[i]),
        sizeOfGrid))) + (sign * 2 * sizeOfGrid);
    }
    else{                     // Co-ordinate is positive
        point[i] = point[i] + (sizeOfGrid - fmod(point[i], sizeOfGrid))
        + (sign * 2 * sizeOfGrid);
    }
}
```

Listing 3: Code snippet showing how minimum and maximum points can be rounded where the sign is negative if minimum and positive otherwise.

Calculating the optimal grid size is important to ensure that the signed distance field produced is an accurate representation of the original mesh. Whilst higher grid resolutions generate more accurate distance fields, they are much slower to compute and result in a highly tessellated isosurface which is slow to simplify. In the opposite case, low resolution grids can miss sharp features in the mesh causing poor isosurfaces to be produced which are not necessarily manifold. Examples of how different grid sizes affect sharp features are shown in Figure 4.1. Given that problems occur at sharp or narrow features, a grid size that is likely to find them is needed. For this implementation, the average triangle edge length is used to estimate the grid size. Using this value makes sharp features likely to be found whilst scaling it for different size models nicely. This value is then multiplied by 1.5 for this implementation to lower the grid size and therefore reduce computation time and isosurface tessellation. Should this value be different from the one the user requires, they can opt to manually input the size of each grid length in which case this calculation will be avoided.

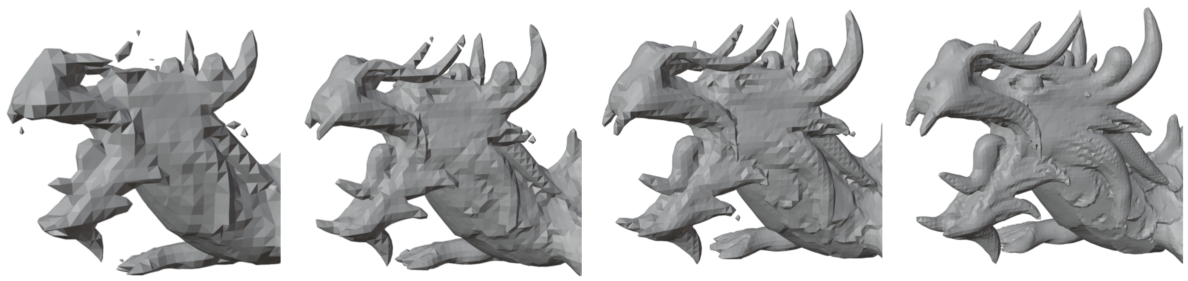


Figure 4.1: Images of a dragon head at varying resolutions of distance fields.

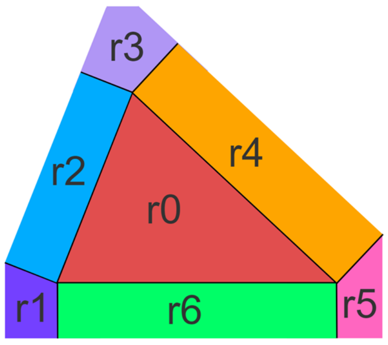


Figure 4.2: A diagram illustrating the 7 different regions possible for a triangle in 2D or 3D space.

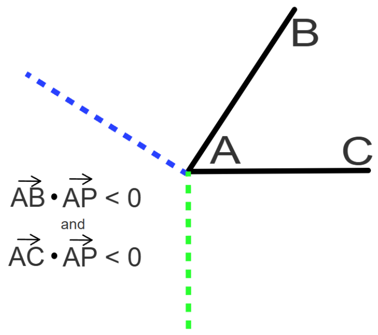


Figure 4.3: Dot product rules for determining if A is the closest vertex.

4.2.2 Distance from point to triangle

To calculate the distance from a grid point P to the closest triangle, the entire mesh must be iterated through per node. Each iteration calculates the distance from the node to the closest point on the current triangle, keeping track of the shortest distance until the entire mesh has been searched. As discussed in the paper by Jones [18], this problem can result in the closest point being on the triangle, on an edge or a vertex. Knowing this, the first step towards determining the closest point is to find the region in which the point lies. A point can lie within 1 of 7 regions on or around the triangle, the regions used for this implementation are largely inspired by the work by Jones and result in the regions shown in Figure 4.2.

The first region determined is r0 which requires P to lie within edges $\overrightarrow{AB}$, $\overrightarrow{BC}$ and $\overrightarrow{CA}$. Considering this, the outwards pointing vector for an edge $\overrightarrow{AB}$ can be found, by taking the cross product of the directed edge and the normal of the triangle. Once the outwards pointing vector is obtained, applying the dot product to that and the vector representing $\overrightarrow{AP}$, returns a negative value if behind the line. The code derived from this concept is shown in the code snippet shown in Listing 4. Should the point lie within r0 after testing all edges of the triangle, calculating the distance to the triangle is simply the distance from the point to the plane. This can be found using equations 1-3 in Jones' paper [18].

```
        float testAB = glm::dot(glm::cross(AB, N), AP);
        float testBC = glm::dot(glm::cross(BC, N), BP);
        float testCA = glm::dot(glm::cross(CA, N), CP);
```

Listing 4: Code snippet showing how the location of P with respect to each edge can be obtained.

Once it can be determined that the point does not lie within the bounds of the triangle, the program aims to find out if P lies closest to a vertex (r1, r3, r5 in Figure 4.2). Similar to the method outlined above, the dot product is used to determine which side of a directed edge P lies 4.3. Using vertex A as an example, A is the closest vertex and $\overrightarrow{AP}$ describes the distance to the triangle if the following Equation 4.1 is true:

$$\overrightarrow{AB} \cdot \overrightarrow{AP} <= 0 \quad \&\& \quad \overrightarrow{AC} \cdot \overrightarrow{AP} <= 0 \tag{4.1}$$

The final regions to test for are the cases where the point P is closest to one of the 3 edges of the triangle (r2, r4, r6). At this stage, the calculations needed to test for this have already been done and it is just a matter of which calculations to use for each case. Determining if $\overrightarrow{AB}$ is the closest edge requires $\overrightarrow{AB} \cdot \overrightarrow{AP}$, $\overrightarrow{BA} \cdot \overrightarrow{BP}$ and the test done to determine which side of $\overrightarrow{AB}$ the point lies (testAB in Listing 4) to all be positive values. Should this be true and the point lies on the edge, calculating the distance can be done as shown in Equation 4.2.

$$length(\frac{(\overrightarrow{AP} \times \overrightarrow{BP})}{length(AB)}) \tag{4.2}$$

4.2.3 Choosing the sign

Once the distances for each point in the scalar field can be calculated, the final step to creating a signed distance field is to find the sign for each distance. The sign of the distance is dependent on which side of the triangle the point lies. For points behind the triangle, which are in the opposite direction to the face normal, they are referred to as positively signed. In the other case, where points are in front of the triangle, distances are negatively signed. As done in the previous section 4.2.2, the dot product can be used to find which side of a plane a point lies in 3D. For this calculation, the vector $\overrightarrow{hP}$, where h is the closest point found on the triangle, must be used with the normal of the triangle at such a point. This requires us to know the normal/pseudonormal [3] for every vertex, edge and triangle in the mesh so that the sign can be accurately produced. To find the normal of each triangle, the cross product, $\overrightarrow{AB} \times \overrightarrow{AC}$, is normalised. Using the face normals, edge normals can be calculated as the normalised sum of the two faces it joins. Finally, vertex normal's can be found as the weighted sum of all triangle normals of which the vertex can be found. The weighting of each face normal is simply the incident angle on the vertex [3] as the resulting normal is normalised. It is worth noting that each of the normals are pre-calculated and stored after the mesh is read into the program to prevent having to re-calculate them on every iteration.

4.2.4 Optimisations

With each node in the grid requiring that all triangles in the mesh be iterated over and the distance calculated, computing the signed distance field for large meshes can be expensive. Fortunately, this algorithm can be easily parallelised to make use of all threads on a CPU, speeding up the implementation significantly. The OpenMP library [31] is used to parallelise the entire set of nested for loops as shown in Listing 5. This splits the work across all available threads on the CPU. For example, using a 12 threaded processor on a grid size of 360x360x360 would result in each thread computing 30x360x360 nodes. This is possible because the only dependencies lie within the nested loop itself, meaning that, should node[30][23][52] complete before node[12][34][16] no data will be overwritten or be relied on by any other calculations.

```
float xStart = minPoint[0];
#pragma omp parallel for
for (int xID = 0; xID < xSize; xID++) {
    float x = xStart + (xID * sizeOfGrid);
    float yStart = minPoint[1];
    for (int yID = 0; yID < ySize; yID++) {
        float y = yStart + (yID * sizeOfGrid);
        float zStart = minPoint[2];
        for (int zID = 0; zID < zSize; zID++) {
            float z = zStart + (zID * sizeOfGrid);
```

Listing 5: Code snippet showing the parallelisation of the grid traversal.

Further optimisations involve the usage of bounding boxes to accelerate the closest triangle search process. This optimisation utilises the idea of bounding volumes mentioned in [19] to reduce the number of triangles checked per grid node. The implementation of this is simple and works especially well for higher distance field resolutions. The first step is to divide the grid into boxes that are much larger than a grid cube, this implementation aims to have each bounding box be around 8 grid lengths in size on each dimension. Once each bounding box is defined, each triangle in the mesh is assigned to one or more bounding boxes. A triangle can be in 3 boxes at most with the position of each vertex on the triangle being the deciding factor. It is important to ensure that bounding boxes do not store multiple references to the same triangle. To prevent this, the unordered set data structure provided in the C++ standard library is used. Once this step is complete, for each node, all bounding boxes within a certain radius are found. The implementation uses a set radius of 14 grid lengths as it is guaranteed to be greater than a single bounding box which is defined as 8x8x8. With the closest bounding boxes found, the triangles contained in those boxes are the closest. This significantly reduces the number of triangles that need to be checked causing a significant speed up.

A frequent scenario of using the bounding box method is that the surrounding bounding boxes have no triangles assigned to them. In this case, no closest triangle can be found so the algorithm can make 1 of 2 choices. The first choice, and safest choice, is to default to checking every triangle in the mesh for the closest triangle. This is slow but produces accurate results in all cases. The second choice employs the idea that there is no nearby triangles and the grid node must be in empty space. Therefore, the algorithm assumes that the surrounding nodes are in the same area of space (outside or inside the mesh) so the exact distance does not matter. In this case, a value of 1000 distance is given to the node and no triangle calculations are made. This results in an inaccurate distance field, however, it yields massive speed ups in SDF calculation time. In order to process the inaccurate distance field, the marching cubes algorithm must take note of any distances of exactly 1000 distance. In the case of a cube containing a single 1000 distance, it can be assumed that the cube is in empty space and is therefore assigned case number 0 resulting in no triangles. This method works extremely well for sparse distance fields where the mesh has lots of inside/outside space. It is however not suitable for meshes with holes in as it cannot guarantee a correct isosurface. This is because the inside and outside of a mesh is not strictly differentiable so points with no nearby triangles may still require the closest triangle to be calculated. To discern the 2 choices, the mesh is checked for holes once the mesh has been read in. If the mesh has no holes, the user will be prompted that the process can be optimised and the second choice is available. Allowing the user to make the choice means that if they wish to use the distance fields in other applications the results can still be valid.

4.3 Marching Cubes Algorithm

To complete the construction of an iso-surface from a given scalar field, such as the one described in the previous section, the marching cubes algorithm detailed by Lorensen and Cline [24] can be applied. This algorithm takes a signed distance field (SDF) and outputs a manifold surface which is to say it is free of holes and self-intersections, as in the aims of the project. It does this by "marching" through each cube of the SDF and generating up to 5 triangles placed at the intersection between the surface and the cube. This section will describe the steps taken to implement this algorithm and the way in which the program can be tested.

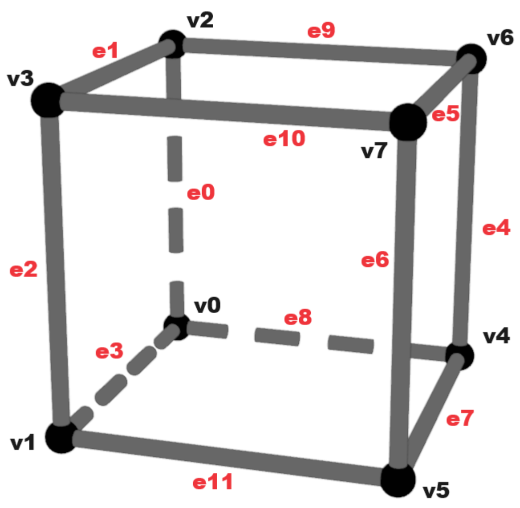


Figure 4.4: A single cube as defined by the look-up table in Appendix A. Vertices are labelled in black and edges in red.

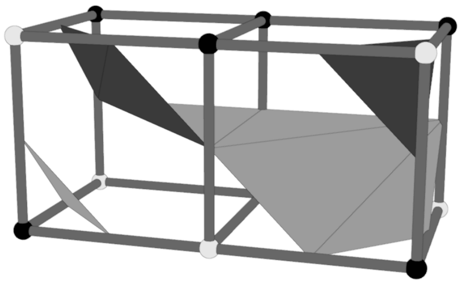
4.3.1 Case tables

For this marching cubes implementation, all 256 cases have to be computed and stored in a look-up table so they can be easily accessed as detailed in the design (3.2.3). As discussed and rectified by Montani et al. [26], the original marching cubes tables [24] contain holes for ambiguous base cases 3, 6, 7, 10, 12 and 13. They therefore proposed that additional cases for the complements of these cases be produced. Applying this to the implementation of this project required amending the original table by Lorensen and Cline with the fixed cases. This subsection will detail the impact of the changes made and the general format the look-up tables take.

To define the look-up table, shown in Figure 4.6, each of the vertices and edges on the cube must be indexed so they can be referred to when placing the vertices of a triangle. The layout of a single cube is shown in Figure 4.4 with vertices labelled 0-7 and edges 0-11. Knowing the indices of each edge on the cube, a single triangle in the look-up table is defined by 3 edge indices. For example, the 3 numbers: 0, 3, 8, signify a triangle with vertices lying on edges 0, 3 and 8 as shown in Case 1 of Figure 4.6. A full example of a case entry can be seen in the below code snippet, where the first number in the array is the number of triangles, the last is the base case (Figure 4.6) and a -1 signifies no more triangles.

```
case[21] = [ 3, 9,1,4, 1,7,4, 7,1,3, -1,-1,-1, -1,-1,-1, 6 ]
```

Before constructing the "fixed" look-up table as proposed in [26], the original table from [24] was tested with the holes being immediately apparent. Cases that should mirror


(a) Using Lorensen and Cline's [24] triangle configurations.

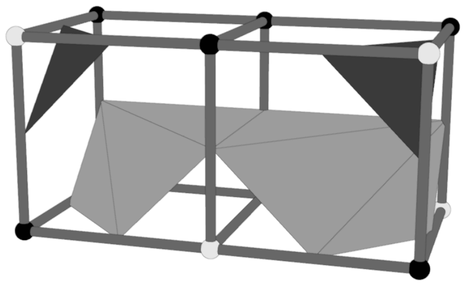

(b) Using Montani et al. [26] triangle configurations.

Figure 4.5: Cases 57 (left cube) and 147 (right cube) side by side. White vertices indicate they are inside the mesh and black outside. Similarly, front facing triangles are light grey and back facing triangles a dark grey.

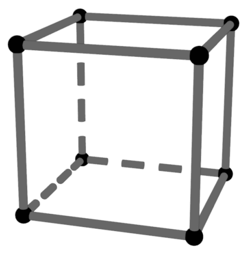
each other when side by side were not always correct, as shown in Figure 4.5a. By checking these cases and noting their configurations before applying the changes by Montani et al., they can be re-tested to check the validity of the finalised look-up tables. Doing just this results in the same configuration used in Figure 4.5a but with a new set of triangles that remove the hole produced by the original tables (see Figure 4.5b). Applying this before and after test to a series of ambiguous cases reveals that the new look-up table produces a watertight mesh every time. Code listing 6 demonstrates how case 57 (the left cube in 4.5) is changed in the look-up table.

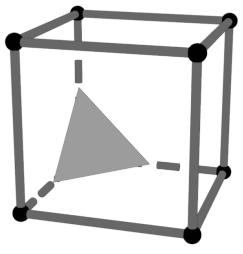
```
// Compliment of case 57, case 198
case[198] = [ 4, 11,3,2, 1,0,10, 0,6,10, 6,0,4, -1,-1,-1, 15 ]
// Case 57 Before changes
case[57]  = [ 4, 11,2,3, 1,10,0, 0,10,6, 6,4,0, -1,-1,-1, 15 ]
// Case 57 After changes
case[57]  = [ 4, 1,10,2, 0,3,11, 0,11,6, 0,6,4, -1,-1,-1, 15 ]
```

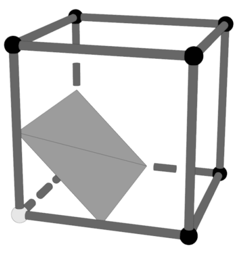
Listing 6: Code detailing the changing of a compliment case as per the paper by Montani et al. [26].

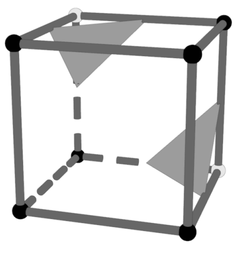
4.3.2 The March

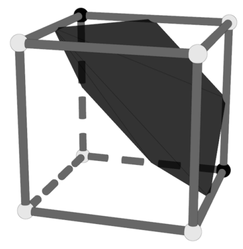
Once the 256 possible cases have been collated into a look-up table (Figure 4.6) that can be referenced in the main application, the main algorithm can begin. For each cubic region of the SDF, the program starts by generating the case number that corresponds to one of the 256 cases in the lookup table. As described in the original marching cubes paper [24], the 256 cases are equivalent to an 8-bit binary number describing the state of each vertex of the cube. Therefore, to calculate the case number, this implementation

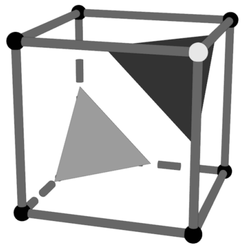

(a) Case 0

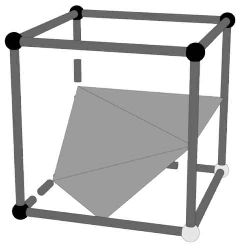

(b) Case 1

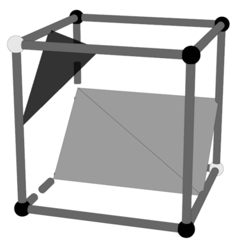

(c) Case 2

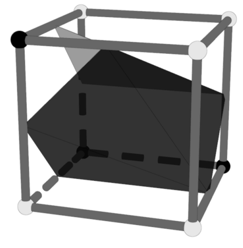

(d) Case 3

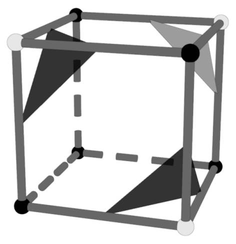

(e) Case 3C

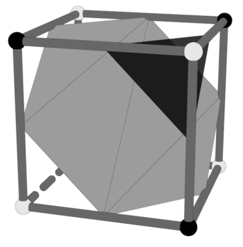

(f) Case 4

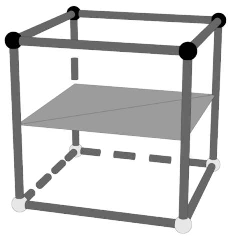

(g) Case 5

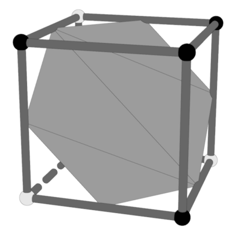

(h) Case 6

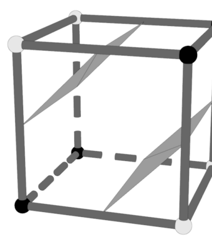

(i) Case 6C

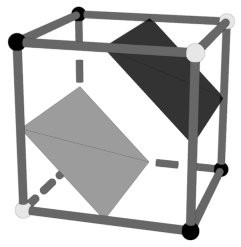

(j) Case 7

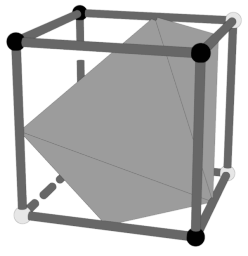

(k) Case 7C

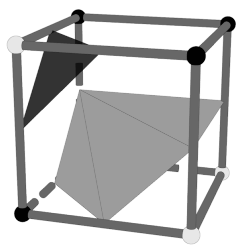

(l) Case 8

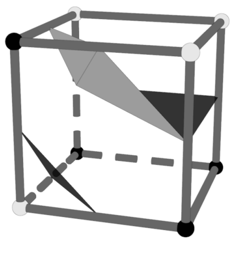

(m) Case 9

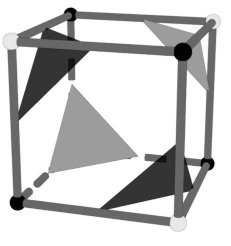

(n) Case 10

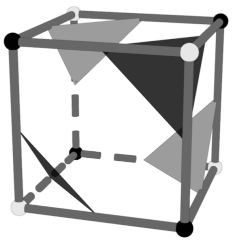

(o) Case 10C


(p) Case 11


(q) Case 12


(r) Case 12C


(s) Case 13


(t) Case 13C

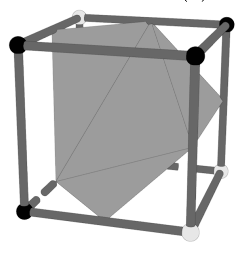

(u) Case 14

Figure 4.6: The set of base cases used for this project was constructed using the original cases by Lorensen and Cline [24] and improved upon using the solution by Montani et al. [26].

traverses each vertex of the cube, in the order outlined in Figure 4.4, applying a 1 if the signed distance is positive and 0 otherwise. This results in case 147 (right hand cube in Figure 4.5b) being represented as 1001 0011 with 4 vertices inside the mesh (1) and 4 outside the mesh (0).

With the case number acquired, the triangles needed to represent the cube are found in the lookup table as a set of vertices placed on the edges of the cube. Each vertex position is found by linearly interpolating along each intersected edge to find its zero crossing as required of the application by FR 2.4. More details on the interpolation algorithm can be found in the next subsection (4.3.3). Each set of 3 vertices defining a triangle within the cube is then pushed back onto an array to be stored. This array, sometimes known as triangle soup, defines the mesh entirely by the end of the algorithm as a set of triangles with no defined order.

To convert from the triangle soup into the directed edge format described in section 4.1.2, the entire array is iterated through with each newly found vertex given a unique vertex ID. This vertex ID is then used to define each face within the structure creating an indexed vertex array and edges array. From this data, the final step is to assign each edge its other half which is done by searching the entire edges array for the occurrence of the inverse edge. An example of this would be the edge defined in the array as $3, 1$ would have its other half where the pattern $1, 3$ occurs. Note that despite the edge array being 1-dimensional each triangle is defined in sets of 3 and therefore edges are circular within that set.

4.3.3 Interpolation of edges

Given a pair of vertices that define an edge and the signed distances corresponding to each vertex of the edge, the intersection point of the surface can be found using linear interpolation [24]. The linear interpolation formula shown in Equation 4.3 can be used to produce the following line of code to retrieve the vertex location of the intersection.

```
vertexA + float(ISOVALUE - distA) * (vertexB - vertexA) / (distB - distA)
```

$$y = y_1 + (x - x_1)\frac{(y_2 - y_1)}{(x_2 - x_1)} \tag{4.3}$$

The implementation above defines ISOVALUE to be 0.00005 which is the point at which the surface crosses the threshold in the SDF. The reason a value of 0.00005 is chosen over an integer value of 0, which would perfectly split negative and positive values, is to

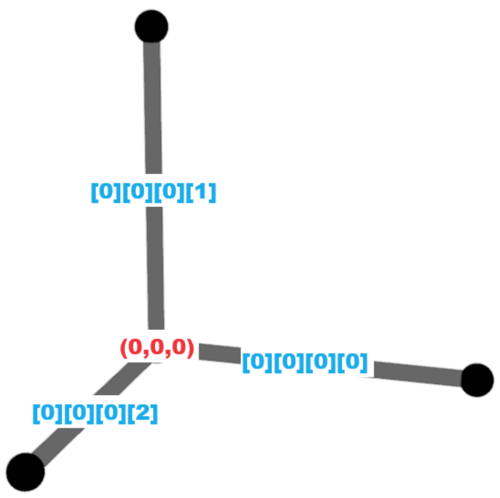


Figure 4.7: Visualisation of entries in 4.1 given vertex position (0,0,0).

prevent the occurrence of zero-area triangles. A zero-area triangle is one where all 3 vertices of the triangle are positioned on the same line or in this case the same vertex. Applying linear interpolation to an edge with distance values 0 and 2 would result in the interpolated point being placed on the vertex whose distance is 0. Whilst this would be entirely correct as that vertex must lie on the surface of the mesh, this would produce the same result for all 3 edges of the triangle. This would create a triangle defined by a single point which can prevent the mesh from meeting the requirements of being manifold. With this knowledge, the floating point precision of the generated SDF, which for this implementation is 4 decimal points, can be exploited to guarantee that the value of 0.00005 could never occur in the SDF. Although this trick results in the reduction of mesh accuracy compared to the original input mesh, the change is trivial when considering the topological correctness of the isosurface.

Another possible error that can result from the interpolation of points along an edge is the result of multiple vertices being placed on the same edge. Due to the possibility of floating point error when calculating with floating point numbers, the same vertex can be stored with differing values. Both vertex values will therefore be registered as unique vertices resulting in a non-manifold mesh as certain directed edges will have no other half. To avoid this, vertices must be limited to one per edge of the SDF, as per FR 2.3, where checks can be done to determine if an edge already has a vertex assigned to it. Furthermore, such checks will reduce the need for re-calculating the same values creating a more efficient and less computationally expensive process. To index this array and store a vertex ID for each edge, the x y and z values define a grid point and assign each grid point 3 edges to keep track of, as shown in Figure 4.7. Using this and the edge IDs assigned with the case tables in 4.4, the look-up table (Table 4.1) can be produced. This allows the fast retrieval of the edge cube array indices once the edge ID within the cube is known. Therefore, on initialisation, the array is filled with -1 values which are replaced with vertex IDs when necessary. This means the implementation only needs to check for the value -1 to see if the edge is empty. It is worth noting that this procedure comes at an increased memory cost with the need for an extra array of size $(xSize - 1) * (ySize - 1) * (zSize - 1) * 3 * 4$ bytes. Whilst this is a considerable amount of memory, especially for very high resolution distance fields, it guarantees that the mesh will not suffer from issues caused by floating point errors.

edge on the cube	xID	yID	zID	edgeID as in Fig 4.4
0	0	0	0	1
1	0	1	0	2
2	0	0	1	1
3	0	0	0	2
4	1	0	0	1
5	1	1	0	2
6	1	0	1	1
7	1	0	0	2
8	0	0	0	0
9	0	1	0	0
10	0	1	1	0
11	0	0	1	0

Table 4.1: The lookup table which takes the edge ID on the cube shown in Figure 4.4 and outputs the set of indices for the grid edge array.

4.4 Edge Collapse

The final process required to generate a low-poly mesh is to reduce the triangle count. In line with the design explained in Section 3.2.4, the edge collapse program takes the generated iso-surface in Section 4.3 and produces a manifold mesh with a significantly reduced polygon count. The program also takes optional inputs from the user regarding the stopping criteria as in FR 3.2 which will directly influence the output. This section will detail the techniques applied to choose an edge to collapse on each iteration as well as the method by which an edge is removed and the mesh simplified.

4.4.1 Quadric Error Metrics

As detailed in subsection 2.2.2, quadric error metrics aim to choose collapsible edges that will have the least effect on the overall shape of the input mesh. Implementation of this cost function closely follows the paper by Garland and Heckbert [13] and their defined algorithm for finding the optimal vertex position.

Finding the quadric

The first step taken, as stated in the paper [13], was to compute a 4x4 matrix for each vertex **Q** which is the sum of squared distances to its planes. Considering this, for each unique vertex **v**, the edge array is searched for each triangle **t** incident on **v**. Upon finding each triangle, the aim is to find the fundamental error quadric **K**, as defined by Garland and Heckbert [13] as $K_p = pp^T$, where p is the vector $[a, b, c, d]$ as defined in Equation 4.4. This requires the equation describing the plane in which **t** lies with Equation 4.4 being used to describe such a plane. Fortunately, values for a, b and c correspond directly to the x, y and z components of the triangle's normal vector which can be easily calculated as the cross product of 2 edges. This makes calculating the final value of d a case of substituting a vertex of the triangle into the equation as x, y and z and re-arranging to find d. The fully implemented function to find K for a given triangle is shown in Listing 7. Matrix Q is then calculated as the sum of all K's belonging to a vertex and stored in a 1D array of 4x4 matrices indexed by vertex ID.

$$ax + by + cz + d = 0 \tag{4.4}$$

```
glm::mat4 findK(int triangleID){
    // Get the triangle
    glm::vec3 A = vertices[faces[triangleID * 3]];
    glm::vec3 B = vertices[faces[triangleID * 3 + 1]];
    glm::vec3 C = vertices[faces[triangleID * 3 + 2]];
    // Get the plane
    // Step 1: Calculate the normal
    glm::vec3 AB = B-A;
    glm::vec3 AC = C-A;
    glm::vec3 norm = glm::normalize(glm::cross(AB, AC));
    // Step 2: Calculate the offset
    float d = -(norm.x * A.x + norm.y * A.y + norm.z * A.z);
    // Step 3: Plane is [a,b,c,d]T
    glm::vec4 p = glm::vec4(norm.x, norm.y, norm.z, d);
    // Step 4: Build the matrix
    glm::mat4 K = glm::outerProduct(p,p);

    return glm::mat4(K);
}
```

Listing 7: Code detailing the calculation of the fundamental error quadric K when given a triangle ID.

For optimisation, it is worth considering that in the paper referenced [13], each matrix Q is symmetric and could be stored as 10 float values rather than the 16 used for the 4x4 matrix. This was ignored for the purpose of this implementation for the sole reason of simplicity and avoiding the reconstruction of matrices at each update step. If this application were to have strict hardware constraints and time requirements this could be improved with minimal extra cost to computation. This is due to the nature of the update step, described in Section 4.4.3, only requiring vertices affected by the collapse to be updated.

Calculating the error

Once $\mathbf{Q}$ is known for all unique vertices, the final step is to calculate an error cost for each directed edge to define the collapse order. Garland and Heckbert [13] define the error of a single collapse to be $\Delta(v) = v^T\mathbf{Q}v$, where v is the vertex that remains after the collapse. It is worth noting that this equation returns an array of 4 values when calculated exactly, which cannot be compared directly. Therefore, the equation is carried out as shown in Equation 4.5 which returns a float value that can be easily used for ordering collapse operations by cost.

$$\Delta(v) = v^T \cdot (\mathbf{Q}v) \tag{4.5}$$

This leaves the question of how v can be chosen for each collapse operation as explained in Subsection 4.4.2. For each directed edge, the resulting quadric $\mathbf{Q}$ is the sum of the two matrices assigned to the vertices on the edge, $\mathbf{Q}_{new} = \mathbf{Q}_{v1} + \mathbf{Q}_{v2}$. Using this, the optimal vertex position, as defined by equation 1 in the paper [13], is calculated as in Equation 4.6 if and only if the determinant of the matrix is not 0.

$$v_{new} = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{4.6}$$

Should the determinant be equal to 0, the application chooses the sending vertex as the optimal choice v_{new} which is stored for each directed edge. This could be improved by choosing the optimal point along the edge in this case, however, due to the time constraints of this project, the simpler solution was chosen. Once the calculations for Q_{new} and v_{new} are complete, they can be used in Equation 4.5 to return the total error cost for the directed edge. It is important to understand that although these calculations can be completed, not every edge collapse results in a topologically valid mesh configuration. To avoid this, further checks must be done before an edge can be ordered on the stack, these checks are detailed in subsection 4.4.3.

4.4.2 The collapse operation

This implementation applies the standard edge collapse algorithm detailed by Hoppe [15], where two vertices are joined across an edge and moved to an optimal position.

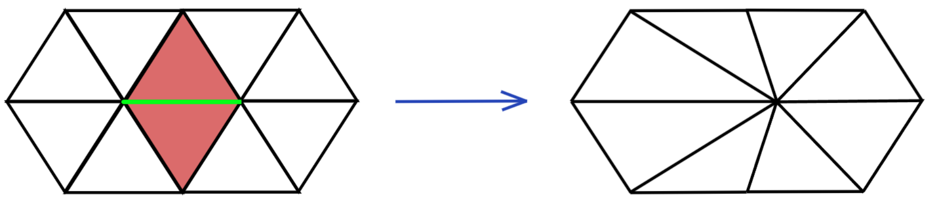


Figure 4.8: Single edge collapse iteration. Collapsing the green edge removes the red faces and results in the mesh on the right.

Each iteration of the edge collapse removes 1 vertex, 2 triangles, and 3 edges from the mesh as seen in Figure 4.8. Assuming that the mesh is topologically correct and that the edge collapse chosen is valid, this operation is Eulerian, which means that it will always result in another manifold mesh. Considering this, it is important that the directed edge data structure be properly maintained during the removal of the data. This means all indices used relating to vertices, edges, and other halves must be updated accordingly when data is moved or deleted from the data structure.

The first removal operation completed is the removal of the 2 faces on either side of the edge. Conveniently, using the directed edge data structure means that the removal of 2 faces and 3 edges can be done with the same operation. Each face is removed one at a time, however, to avoid holes in the data and mesh, only the last 3 edges/faces in the array can be removed safely. Therefore, if it is the case that the face to be removed is not the last 3 entries of the array it is swapped with the final 3 entries. Whilst this then allows the face to be removed, it creates a problem within the other half array as it is indexed by the edge position. To fix this the other halves are swapped as well with the triangle being sent to the back discarded. This is because only the other halves belonging to the kept triangle need swapping. The update of the other half array following the face swap is best outlined by the implemented code shown in Listing 8. In the case that the face to be removed is already at the back of the array, no swap occurs and instead the other halves for each edge of the triangle are assigned a -1 or -2 value. A -1 is assigned if it is the first triangle to be removed and -2 for the second. This is necessary for the re-joining of other halves once the face removal is complete.

Once the faces are removed successfully, the final step is to iterate through the entire edge array replacing all occurrences of the removed vertex with the index of the kept vertex. This implementation chooses to keep the sending vertex's index and update its position to the optimal one stored with the edge ID as detailed in Subsection 4.4.1. This joins the hole created by the removal of both faces producing a complete mesh free of holes. Whilst this would suffice for exporting the file, the other half array must be updated so the next iteration can function properly. This is where the -1 and -2 values given to the other halves during the removal operation become helpful. Whilst searching

```
// For each edge in the triangle
for(int i = 0; i < 3; i++){
    // If the edge will be the same but flipped remove the edge
    if(faceID+i == otherHalf[otherHalf.size()-(3-i)]){
        otherHalf[faceID+i] = -1 * faceNum;
    }
    else{
        // If the edge already has no other half
        if(otherHalf[faceID+i] != -1){
            otherHalf[otherHalf[faceID+i]] = -1 * faceNum;
        }
        // Swap the other half
        otherHalf[faceID+i] = otherHalf[otherHalf.size()-(3-i)];
        // Update the corresponding other half with the new position
        otherHalf[otherHalf[faceID+i]] = faceID+i;
    }
}
```

Listing 8: The code used to update the other half array after a face has been swapped from the back of the array.

the entire edge array to re-assign the vertices, each other half can be checked for occurrences of -1 or -2. If found, the edgeID is added to a tuple, since there will be exactly 2 of each flag. Each tuple can then be used to assign each other's edgeID as the other half. This completes the removal operation and maintains the data structure entirely. A code snippet detailing the joining of other halves can be seen in Listing 9. To avoid updating the entire edge array per iteration, vertices are removed in bulk at the end of the process before outputting to the .OBJ file format. This is because the edge array should be significantly smaller by then and require less computation to apply a cascading update to remove vertex IDs.

4.4.3 Update and Validation checks

Each iteration of the edge collapse requires the error cost for all affected edges to be updated and the order of collapse operations to be re-sorted. Furthermore, whilst calculating the error cost, a collapse operation must be assessed for validity and omitted from the collapse ordering entirely if invalid.

Update Step

To avoid doing unnecessary computation and speed up the collapse process significantly, each updated edge during the face removal process is inserted into an unordered set. In addition to edges that have been directly modified, it is possible that any edges within 2 neighbourhoods or the two-ring of the remaining vertex can be invalid or have differing

```
// For each edge in the triangle
for(size_t eID = 0; eID < faces.size(); eID++){
    // Check for removed half edge on the first removed face
    if(otherHalf[eID] == -1){
        firstEdge.push_back(eID);
    }
    else if(otherHalf[eID] == -2){
        secondEdge.push_back(eID);
    }
}
// Update the other halves for the kept edges of the removed triangles
otherHalf[firstEdge[0]] = firstEdge[1];
otherHalf[firstEdge[1]] = firstEdge[0];
otherHalf[secondEdge[0]] = secondEdge[1];
otherHalf[secondEdge[1]] = secondEdge[0];
```

Listing 9: A code snippet used to join the other halves from the removed triangles.

error costs. Accounting for this, each vertex within the one-ring has its edge IDs inserted into the set of edges requiring an update. Whilst this additional set of checks increases computational overhead, avoiding this step can cause invalid edges to be collapsed, resulting in holes in the mesh or flipped triangles. It is also true that for large meshes, re-calculating all error costs is extremely costly, so minimising the number of edges updated is optimal.

4.4.4 Validation Step

For this implementation, there are 2 cases in which an edge collapse is invalid and must be omitted from the collapse order to prevent non-manifold outcomes. The first scenario, shown in Figure 4.9 and outlined on page 119 of [6], results in overlapping triangles. To avoid this, the one-rings of each vertex defining the edge must intersect exactly twice. If it is the case that 3 or more intersections are found, the update can be avoided and the error cost stored as -1 so it can be left from the collapse order. Equally, 2 intersections when both one-rings have only 3 vertices, suggest a tetrahedron is present which cannot be decimated further, resulting in another invalid case. Calculating the one-ring twice per error update can be costly as it requires iterating through the entire edge array. To avoid this, the one-ring is calculated once for each vertex at the start of the program. This can then be referenced for each vertex whenever it is needed, providing that it is kept up to date with each collapse. The update step for the one-ring array is then simply the union of the one-ring around each vertex on the edge - excluding the kept vertex and removed vertex IDs. It is also necessary for the removed vertex to be erased from all vertices in its one ring and the kept vertex added if it is not already present.

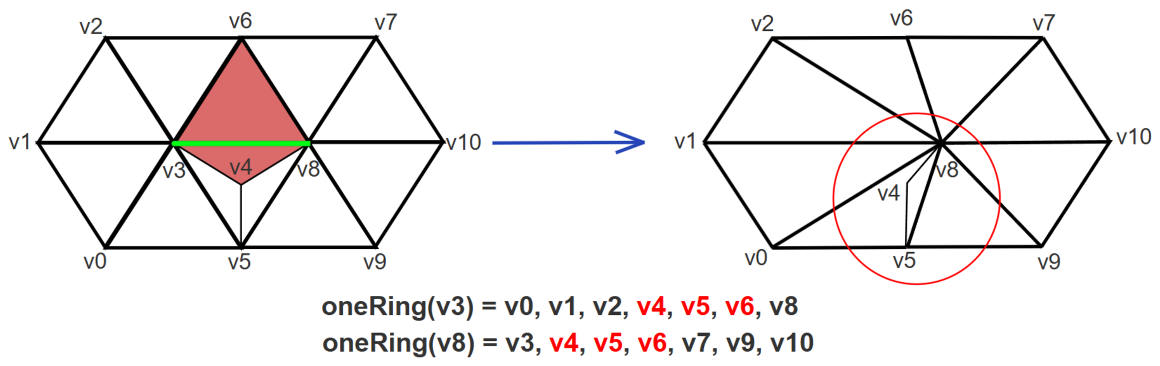

v2
v6
v7
v1
v3
v4
v8
v10
v0
v5
v9
v2
v6
v7
v1
v4
v8
v10
v0
v5
v9
oneRing(v3) = v0, v1, v2, v4, v5, v6, v8
oneRing(v8) = v3, v4, v5, v6, v7, v9, v10

Figure 4.9: Invalid edge collapse operation resulting in overlapping triangles and invalid topology.

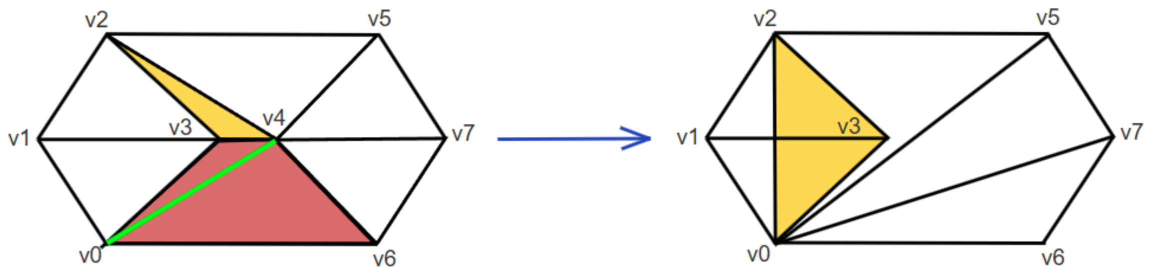


Figure 4.10: Invalid edge collapse operation resulting in a flipped triangle causing self-intersection.

The second case results in a flipped triangle which causes overlaps in the mesh as shown in Figure 4.10. Once the optimal vertex for the collapse has been calculated, each triangle that would inherit the new vertex compares its current normal to the normal it would have should the old vertex be replaced. If the dot product between the old normal and the new normal is less than 0 the triangle has been flipped and is therefore an invalid collapse. Once again this is signalled with a -1 in place of the error cost and the error cost update can exit early. This check can be costly due to the computation of the normal twice per triangle. It is possible to store the normal for each triangle and therefore only require one cross product per check at the cost of added memory overhead. There is also a special invalid case that can be found during this check that whilst not immediately noticeable, can cause holes to occur in further collapses. The case in question is where a zero-area triangle is made. This triangle then has a normal of (0,0,0) which returns 0 for the dot-product calculation used to check for the normal flip. Therefore, to avoid this case, the less than or equal to operator is used to assess the validity of the dot product calculation.

4.4.5 The full algorithm and test cases

Combining the methods detailed in the above subsections results in the iterative reduction of the given isosurface as detailed in the design outlined in Section 3.2.4.

Whilst the specifics are detailed for each key component of the entire algorithm, the ordering and final steps before exporting to OBJ are summarised in the following algorithm, Listing 10.

```
for each vertex
    find error quadric Q
for each edge
    find error cost
sort error list
while (error list != empty) || (numTriangles < desiredTriangleCount)
    if (smallest error > errorTolerance) exit loop
    get keptVertexID and removedVertexID
    remove faces
    replace all instances of removedVertexID with keptVertexID
    for each updated edge
        find error cost
    sort error list
for each removed vertex
    remove vertex from vertices
    decrement all vertex IDs greater than removed ID
for each vertex
    calculate vertex normal
```

Listing 10: The overall edge collapse algorithm.

The main loop of the application is stopped if one of 3 criteria is met. The first and only non-optional criterion is that there are no more valid edges to collapse. This prevents the output mesh from becoming non-manifold by being forced to collapse edges that would result in holes or triangle folds. The other 2 criteria can have optional values, these being the number of triangles to reach and the error tolerance. Should the user wish to specify one and not the other the code will only use that stopping criteria along with the valid edge check. Specifying both criteria will stop the program when the first one is met. In the case that neither one of the criteria is specified, the error tolerance is defaulted to 5.0. This is to prevent the mesh from becoming too dissimilar to the original input in the default use case.

To confirm that the collapse algorithm works as intended, some simple test cases were produced with known and calculable outcomes. Each test case was based on the reduction of an octagon as shown in Figure 4.11 where the outcome of the first collapse results in a pyramid being produced, see Figure 4.12. By discarding the quadric error calculation step and specifying the edge to collapse, the face removal operation can be tested explicitly. Furthermore, due to the small size of the mesh, the data structure is easily readable and can be debugged with hand calculations to ensure the correct

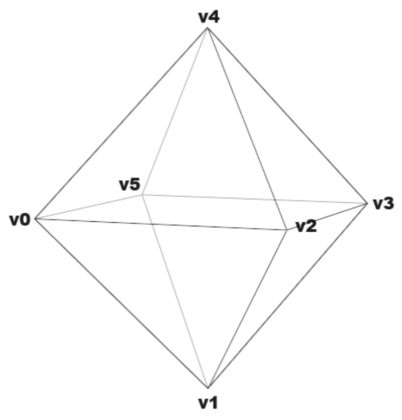


Figure 4.11: Octagon test case before collapsing edges.

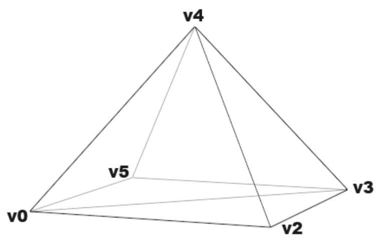


Figure 4.12: Octagon test case with after collapsing edge 0 (v0 to v1).

updates are made. See Tables 4.2 and 4.3 for the result of a singular collapse operation on the octagon and the resulting data structures (without vertex removal step). A further collapse of the pyramid will result in either a plane or a tetrahedron. By ensuring that both iterations work as expected here, any holes created by the full algorithm are likely due to validity checks being missed and invalid edges being collapsed.

edgeID	0	1	2	3	4	5	6	7	8	9	10	11
vertexID	0	1	2	2	1	3	4	0	2	4	2	3
otherHalfID	13	3	7	1	15	10	20	2	9	8	5	21
edgeID	12	13	14	15	16	17	18	19	20	21	22	23
vertexID	5	1	0	3	1	5	4	5	0	4	3	5
otherHalfID	16	0	19	4	12	22	23	14	6	11	17	18

Table 4.2: The state of the edge and other half array after before any edge collapses on the octagon shown in 4.11.

edgeID	0	1	2	3	4	5	6	7	8	9	10	11
vertexID	4	3	5	2	0	3	4	0	2	4	2	3
otherHalfID	11	17	12	7	15	10	14	3	9	8	5	0
edgeID	12	13	14	15	16	17	18	19	20	21	22	23
vertexID	4	5	0	3	0	5	-	-	-	-	-	-
otherHalfID	2	16	6	4	13	1	-	-	-	-	-	-

Table 4.3: The state of the edge and other half array after collapsing edge 0 (v0 to v1). Changed values are highlighted in red with removed values denoted by a dash. 4.11.

Chapter 5

Software Testing and Evaluation

Considering the outcome of the project with regards to the project aims (Section 1.1) and requirements (section 3.1), is essential to evaluating the implementation outlined in Chapter 4. The evaluation techniques used are designed to assess non-functional requirements 1, 4 and 5. Furthermore, the set of criteria evaluated has been chosen to align with other research papers on low-poly mesh generation such as the ones mentioned in the literature review (section 2.1). In particular, the papers by Gao et al. [12] and Chen et al. [8] present varied evaluation methods. This chapter will detail such evaluation methods, explaining how they were carried out, how metrics were obtained and how the results compare against tools used in industry.

To maintain fairness in the results, each section will evaluate the same 9 input meshes displayed in Figure 5.1. This will allow results to be discussed comparatively in the concluding statements (Chapter 6). To keep results representative of the program, optional user inputs (as shown in Figure 3.1) will not be used. Instead, the default conditions set out in the implementations will be used so that no unintended bias can be introduced which may favour different test cases. For example, choosing a large number of desired triangles would result in very fast outputs for lower poly inputs whilst larger meshes would have minimal change. By allowing the program to use default settings, the results will be a good representation of the program's performance without human intervention. In addition to this, default settings allow non-functional requirements 1 and 5 to be tested with only the program's capabilities.

5.1 Test cases

To test the program 9 different object files will be used each with varying polygon counts and topological qualities. Test cases were obtained through freely available online resources such as the GitHub repository of common 3d test cases [17] and the Thingi10k dataset [41] which links to Thingiverse [38] which is a model sharing website for 3D printing. Each case was chosen to provide a set of manifold and non-manifold meshes with the minimum number of polygons wanted being 10,000. This is so the full extent of the project requirements can be assessed, particularly NFR 5. Furthermore, the largest mesh tested will be no more than 250k due to time constraints and the hardware making very large meshes unfeasible to test. Each of the test cases used for this evaluation is shown in Figure 5.1 with details of their topology in Table 5.1 and sources listed in Table 5.2.

Mesh ID	Mesh Name	No. Triangles	Hole free	Is Self-intersecting
1	Armadillo	99,976	True	False
2	Buddha	98,601	False	False
3	Bunny	69,451	False	False
4	Dragon	249,882	True	False
5	Egg Cup	56,722	True	True
6	Monster	69,950	True	False
7	Rocket	47,032	True	False
8	Squirrel	17,596	True	False
9	T.Rex	167,766	False	True

Table 5.1: Table showing the set of test cases used to evaluate the project.

Mesh ID	Original Name	Source	Original Source
1	Armadillo	[17]	Stanford University Computer Graphics Laboratory
2	Happy Buddha	[17]	Stanford University Computer Graphics Laboratory
3	Stanford Bunny	[17]	Stanford University Computer Graphics Laboratory
4	xyzrgb dragon	[17]	Stanford University Computer Graphics Laboratory / `https://www.xyzrgb.com/`
5	Egg cup	[38]	`https://www.thingiverse.com/thing:17079`
6	Monster	[38]	`https://www.thingiverse.com/thing:30415`
7	Rocket Retro 004	[38]	`https://www.thingiverse.com/thing:26163`
8	Squirrel	[38]	`https://www.thingiverse.com/thing:11705`
9	T-Rex Skull	[38]	`https://www.thingiverse.com/thing:308335`

Table 5.2: Table showing the set of test cases and where their original sources and sources can be found.

5.2 Manifold output tests

As a detailed requirement for the project (FR 3.5, NFR 4), all output meshes generated by the program should be manifold. This means that meshes must be watertight, include no pinch points, and be free of self-intersection. Due to time constraints and the complexity involved with checking for self-intersection, this evaluation will only check for pinch points and holes in the mesh. This section will detail the testing process for each criterion and the results acquired from each of the test cases detailed in Section 5.1. Furthermore, the source code for the manifold checker can be found in the GitHub

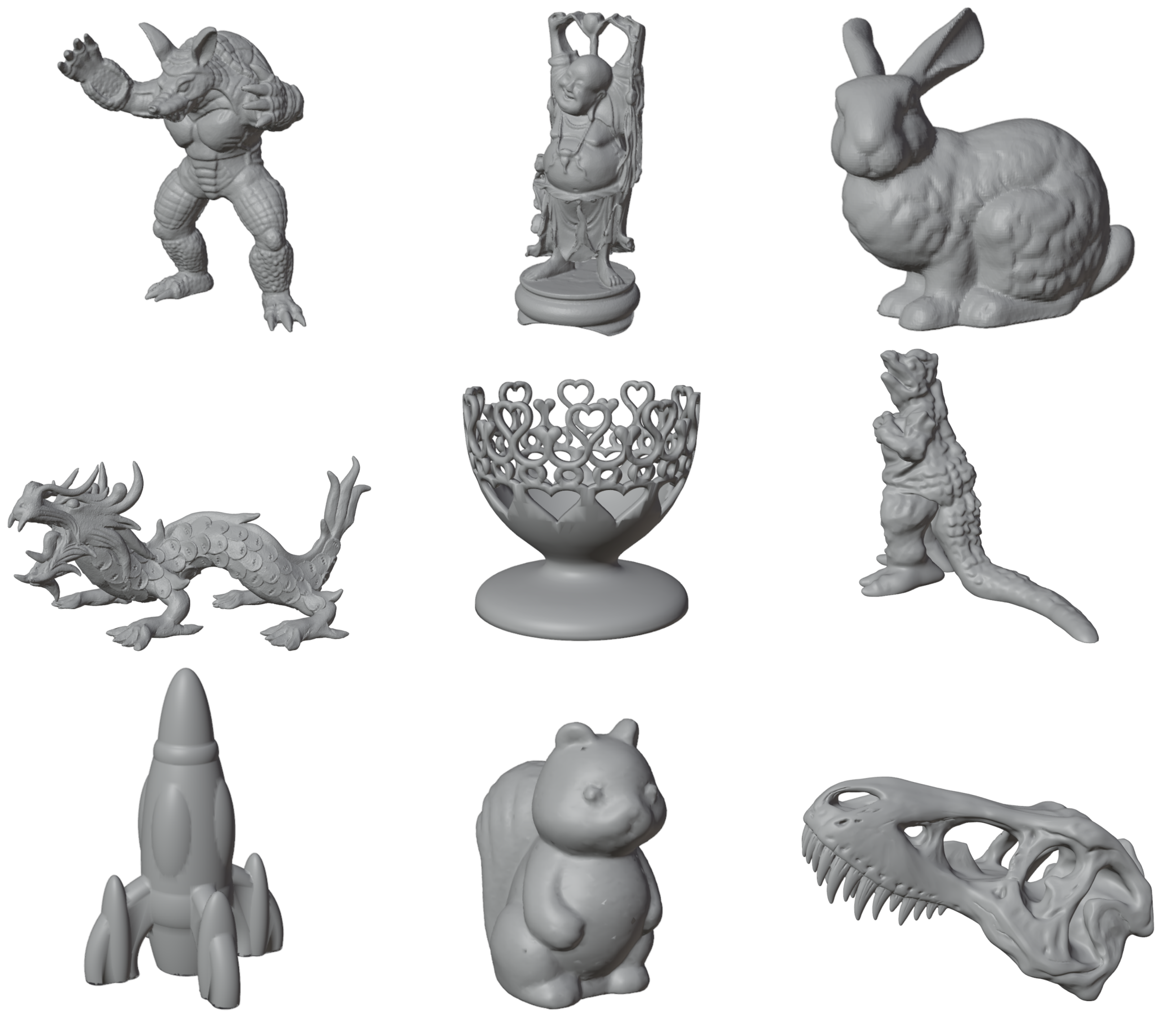


Figure 5.1: Test cases to be used throughout the evaluation process.

repository linked in Appendix A as `ManifoldChecker.cpp`. Testing a mesh to see if it is manifold is then simply a case of inputting it into the program and waiting for the result.

5.2.1 Testing for holes

Using the directed edge data structure [39] as implemented in 4.1.2, testing for holes is quite simple. A requirement for a manifold mesh is that all directed edges must occur in pairs. Therefore, each edge must have a corresponding other half which works both ways. For example, if edge 5 has other half 11 then edge 11 must have other half 5. Therefore, the other half array can be exploited to check for this case. Should any cases be found that do not meet this requirement then it can be determined that the mesh is non-manifold as the offending edge must have a hole/border on one side.

5.2.2 Testing for pinch points

A pinch point is where separate components of a mesh are joined by only a single vertex. This results in multiple cycles possible around a vertex making it non-manifold.

Testing for pinch points is more complex than testing for holes as there are no immediate ambiguities in the data structure. To test for this, the mesh must be entirely traversed from a single starting point to determine if all faces can be reached. If it is the case that not all faces can be reached, the mesh must either contain pinch points or consist of multiple components. In either case, the mesh would not be manifold and would result in the mesh being marked as non-manifold in the results table TABLE.

5.2.3 The results

Inputting the evaluation test cases (Table 5.1) into the manifold checking application yielded the results displayed in Table 5.3. The table has been split into the various criteria required for a mesh to be manifold so that any patterns of failure can be noted.

Mesh ID	Watertight	Pinch-point Free	Singular Component	Was Manifold	Is Manifold
1	True	True	True	True	True
2	True	True	False	False	False
3	True	True	True	False	False
4	True	True	True	True	True
5	True	True	True	False	True
6	True	True	True	True	True
7	True	True	True	True	True
8	True	True	True	True	True
9	True	True	False	False	False

Table 5.3: Table showing the results of the manifold testing on each of the test cases shown in Table 5.1.

The results table shows that 6 of the 9 test cases resulted in manifold outcomes as per the results of the manifold checker. In the 3 cases where the outputs aren't manifold, they all fail to be a single polyhedron, this is represented by the singular component column in 5.3. Observing the state of the failing meshes shows that none were completely manifold before due to all meshes having holes and mesh 9 having self-intersections. Looking at the manifold outputs, 1 of the 4 non-manifold inputs has been fully repaired with all of the manifold inputs being maintained.

5.3 Computation times

This project aims to compute a low-poly mesh within a reasonable time frame as stated in section 1.1. The aim is then further refined into non-functional requirement 1, which defines a minimum criteria for the application to meet in terms of processing time. It is therefore necessary for the execution time of the program to be measured so that it can be assessed. This section of the evaluation will measure the time taken to execute on each of the test cases (section 5.1) and comment on the results in terms of the hardware and input parameters.

Due to the separation of the applications, as discussed in section 3.2.1, the total time for the application cannot be measured in one go. For this reason, execution time was measured for each program separately and then totalled up to represent the complete execution time of the entire process. To ensure that the results were accurate, timers were incorporated into the code using the *chrono* library, where the time at the start of the application is compared to the time at the end and a run time is calculated. The start time is taken as soon as the input files have been read into memory and user inputs are taken. The final time is taken before outputting the data to file once all calculations are done. The reason that the timer starts once the inputs are taken is to prevent the timer from giving bad results due to delayed input from the user. Despite the timer not encapsulating the full program, the main computation of the code is covered so the results should still be representative of which application is taking the longest.

5.3.1 Hardware and Testing Environment

To keep the testing of each test case fair, all executions were completed on the same machine with minimal background applications running. This ensures that all timings can be compared against each other without additional considerations needed for different scenarios or hardware. The system specifications for the machine used for all testing are detailed in Table 5.4. More details on the CPU of the system are available in Table 5.5 since the programs are heavily CPU reliant. Details of the GPU specifics have been omitted as it is not used with the current implementation.

Operating System	Windows 11 Pro N
CPU	AMD Ryzen 5 5600X
RAM	16GB / DDR4 / 2400MHz
GPU	NVIDIA GeForce RTX 3070 / 8GB GDDR6
Storage	1TB M.2 NVMe PCIe Internal SSD

Table 5.4: System specifications of testing machine.

CPU	AMD Ryzen 5 5600X
No. Cores	6
No. Threads	12
Base Clock	3.7GHz
Max Clock	4.6GHz
Cache	32MB

Table 5.5: Specifications of the AMD Ryzen 5 5600x CPU.

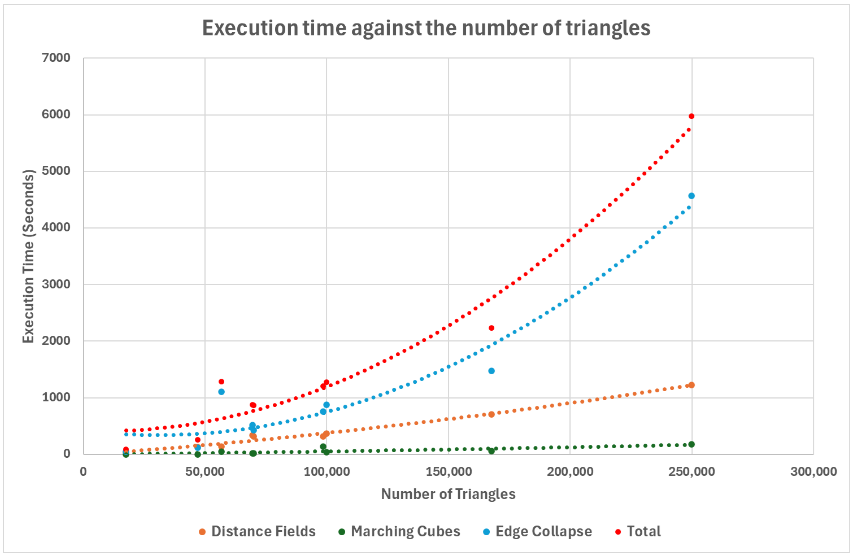


Figure 5.2: A scatter plot showing the execution timings shown in Table 5.6 against the input number of triangles.

5.3.2 The results

Measuring the computation time across all 3 stages of the low-poly mesh generation program produces the results table, 5.6.

Mesh ID	Distance Fields	Marching Cubes	Edge Collapse	Total Time
1	364	34	876	1274
2	316	135	756	1207
3	341	18	519	876
4	1229	182	4564	5975
5	135	46	1104	1285
6	315	14	423	866
7	129	3	121	253
8	35	1	51	87
9	706	56	1470	2232

Table 5.6: Table showing the full process execution times for each of the test cases shown in Table 5.1. Time is shown in seconds.

The results table 5.6 shows the execution time for each of the 3 stages of the low poly mesh generation. Plotting these results on a graph of execution time against the number of triangles produces Figure 5.2. This visualisation clearly highlights which application takes the most time, this being the edge collapse (light blue). The second most time consuming application is the distance fields with the marching cubes being the least. Further looking at the data, time is seen to increase linearly with the marching cubes and distance fields algorithms whilst it increases exponentially with the edge collapse. It can be therefore assumed that larger triangle counts would only increase the execution time further and at a faster rate. Considering the outliers on the data, the egg cup (mesh ID 5) takes considerably longer to calculate for the edge collapse than what the trend suggests it should take. Furthermore, mesh 9 operates much faster than the trend line would suggest, which suggests that the execution time increase between 160k triangles and 250k triangles is larger than suggested by the lines of best fit.

5.4 Polygon reduction

The main aim of this project is to reduce the number of polygons from the original mesh as stated in NFR 5. It is therefore important to test this by evaluating the number of triangles before and after the application is used. To produce results for this evaluation, the number of polygons, as shown in Table 5.1, is taken and compared against the number of polygons in the final output. To provide further insight into the workings of the application, the number of polygons in the produced isosurface will also be shown. These values will be framed in the context of percentage reduction so they can be directly compared with NFR 5 to assess the success of this project given its aims. It is important to note that, as stated in the opening of this chapter, all models were run with default parameters to ensure fairness. Therefore, each model shows the number of polygons reached when the cheapest edge collapse had an error cost greater than 5.

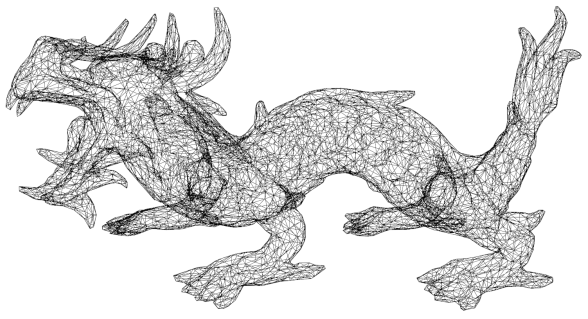
5.4.1 The results

The results of the polygon reduction evaluation are shown in Table 5.7 and show the percentage of original triangles remaining in the output mesh.

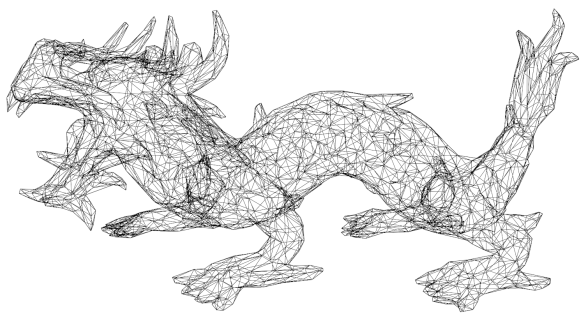
Mesh ID	Input No. Triangles	Isosurface No. Triangles	Output No. Triangles	Percentage Reduction
1	99,976	48,776	5,264	94.73%
2	98,601	44,256	3,628	96.32%
3	69,451	35,744	2,204	96.83%
4	249,882	117,336	11,546	95.38%
5	56,722	57,472	4,412	92.22%
6	69,950	31,556	2,372	96.61%
7	47,032	14,504	1,120	97.62%
8	17,596	7,344	446	97.47%
9	167,766	63,776	3,976	97.63%

Table 5.7: Table showing the triangle reduction of each test case shown in Table 5.1.

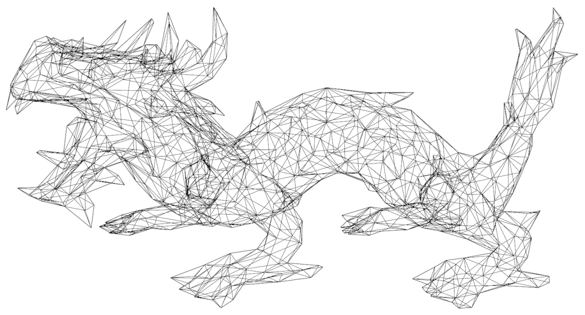
The table shows a minimum reduction of 92.22% and a maximum reduction of 97.63% when stopping at an error cost of 5. These results appear to correlate with the reduction produced by converting the input mesh to an isosurface. In all except mesh 5, the input is reduced by around 50% before any simplification is even done. Mesh 5 seems to show the opposite effect with the number of triangles increasing with the conversion to an



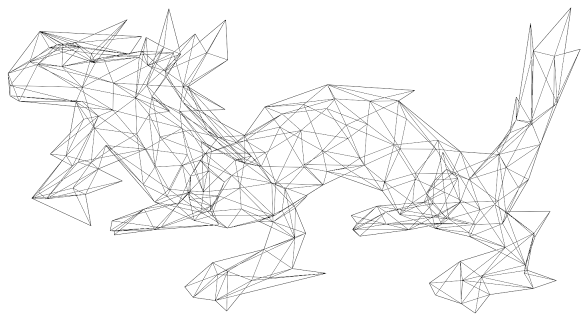
(a) Dragon with 11546 triangles.



(b) Dragon with 5000 triangles.



(c) Dragon with 2000 triangles.



(d) Dragon with 500 triangles.

Figure 5.3: Wire-frame images of the dragon mesh (mesh 4) at different polygon counts demonstrating the usage of the defining triangle count input.

isosurface. This could be the reason why the total reduction is the highest compared to the other cases.

It is also worth testing that the program works when specifying a desired triangle count to achieve. In the below images, Figure 5.3 shows mesh 4 at lower triangulations than the one achieved with the default stopping criteria. The images show that the user can reduce the number of polygons much further than the default allows in most cases. Furthermore, by re-feeding the result of the default run into the simplification algorithm, the time taken the produce the output can be avoided.

5.5 Visual similarity

The visual similarity of 2 meshes can be evaluated quantitatively or qualitatively due to it being how the mesh looks rendered. Evaluating NFR 7 quantitatively would require another program capable of generating an error metric between the two methods. Examples of these techniques are used by Chen et al. [8] to evaluate their method and consist of the Hausdorff distance, Light field distance, Silhouette and normal differences, and Peak signal to noise ratio. Each of these techniques produces a similarity metric that can be used to determine how similar 2 meshes are. Contrasting from this, the qualitative method is much simpler and consists of comparing the two meshes side by side and making a statement as to their visual similarity. Whilst numerical methods like

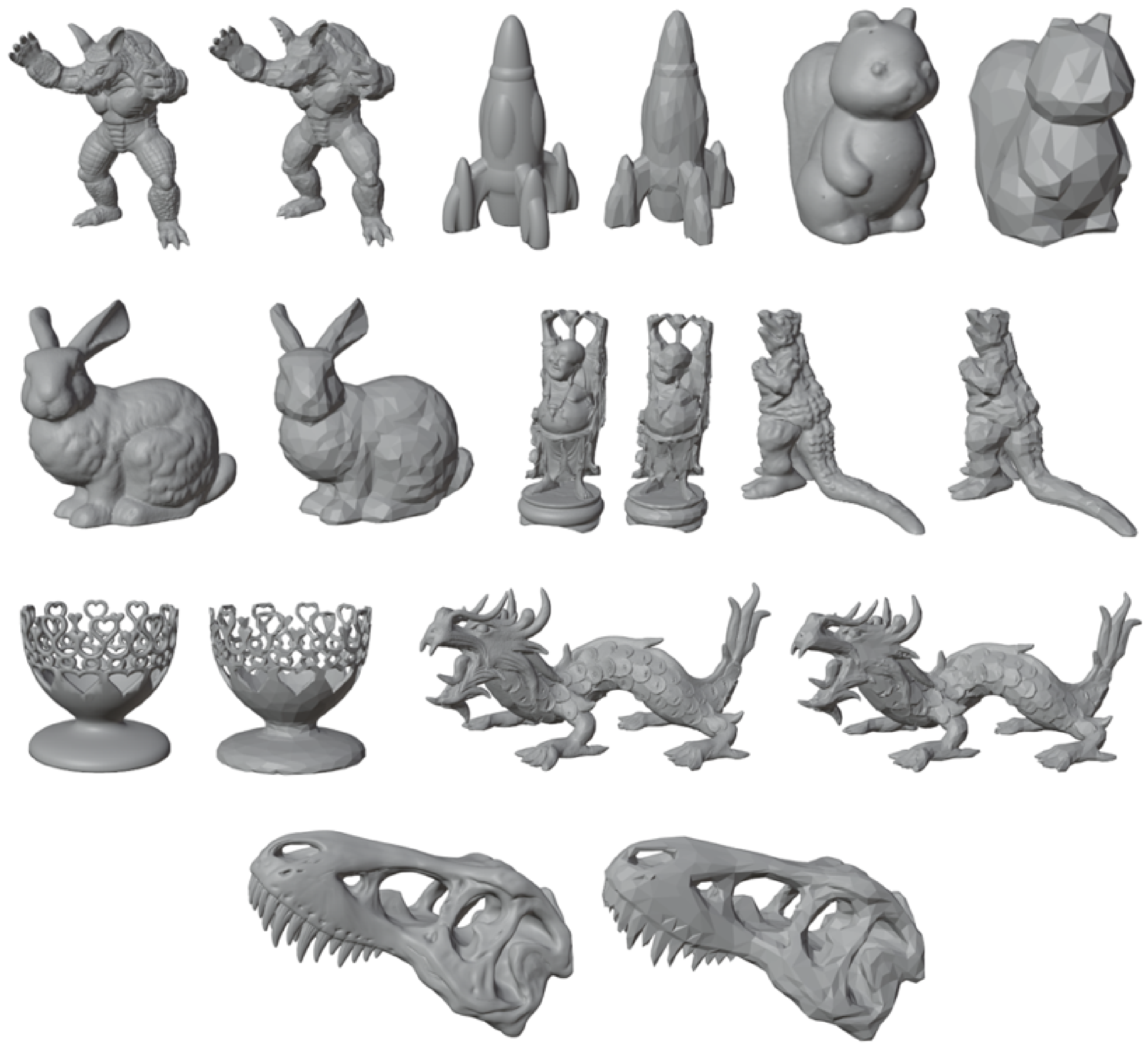


Figure 5.4: All before and after images of test cases (as described in Table 5.1) with input images on the left and low-poly output on the right. Meshes have been rendered using Blender [5].

the Hausdorff Distance [22] allow more accurate comparisons and assumptions to be made implementing a program to do this takes time. Therefore, due to the time constraints and the scope of this project, the qualitative method will be used for this evaluation. This is acceptable since considering the context of the project, the output will likely be used in games where visual quality to the eye is what matters.

5.5.1 The results

To carry out the visual comparison method, before and after images of the mesh are gathered and compared side by side as shown in Figure 5.4. The results of this comparison show very little deviation in shape or features. This is especially true for the larger meshes, these being the dragon (mesh 4) and the T.Rex skull (mesh 9). Smaller models such as the squirrel have their reductions more noticeable, whilst the general shape is preserved, the distinct lack of triangles stands out. Further inspecting the

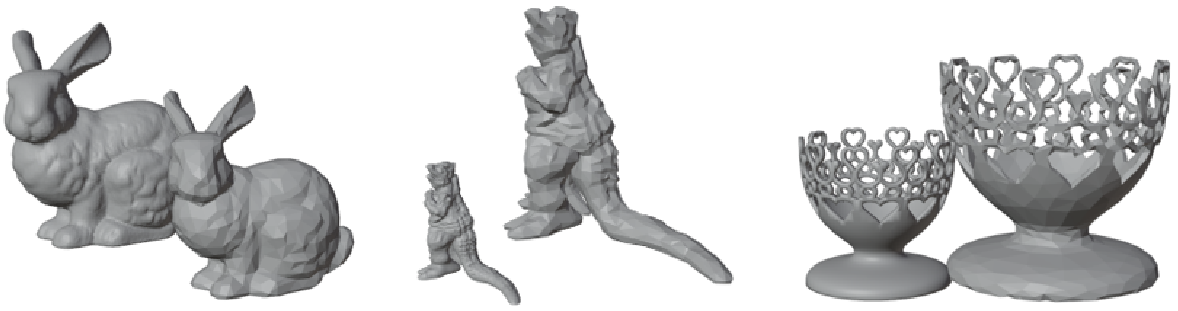


Figure 5.5: Size comparison of meshes 3, 5, and 6 with input meshes on the left and output meshes on the right.

visual properties of the output meshes, it can be noticed that the scale of the input mesh is not retained through the process. This can be seen in the scale comparison images in Figure 5.5. Whilst the rabbits are of similar size, the monster and egg cup are considerably larger on the output than the input. Therefore, it must be noted that whilst features and shape are preserved, the scale, whilst proportionate, can be of different size to the input mesh. This is something that will be discussed in section 5.7 as to whether this is acceptable as well as possible causes for the problem.

5.6 Comparison with industrial tools

This section will compare the results of this project with the ones produced by similar applications in the industry. This stage of the evaluation will focus on how the outputs compare visually and topologically as well as the time taken to produce each result. The applications used are the ones produced by Chen et al. [8] as well as the inbuilt tools in Blender [5]. Whilst other applications are used frequently in industry and other academic evaluations [12, 8] like Simplygon [37], they are not freely available so could not be used in this project. Furthermore, the program by Gao et al. [12] was considered for comparison. However, it was discarded due to the application being made for building models only, meaning any comparison to it would have been biased towards the model produced for this project.

The 2 programs are given the same 9 test cases (Table 5.1) and used to simplify the mesh. The results of the program produced by Chen et al. are shown in Table 5.8 with some visual examples of the outputs in Figure 5.6. Blender allows meshes to be reduced to any ratio of triangles in the input mesh. Furthermore, this is done in extremely fast time (less than 2 seconds), therefore, only visual examples of the output meshes are shown in Figure 5.6 with meshes produced to 5% of the original polygon count. This is so the visual comparison with the ones produced by this report can be most similar.

Mesh ID	Execution Time	Manifold Output	No. Triangles
1	20	True	158
2	24	True	142
3	33	True	406
4	2	True	249
5	34	True	534
6	19	True	146
7	16	True	156
8	22	True	122
9	31	True	238

Table 5.8: Table showing the results of the Robust low poly meshing application by Chen et al. [8].

The results of Table 5.8 show that the application performs much faster than the one produced by this project and results in much fewer triangles. Furthermore, running `ManifoldChecker.cpp` results in all of the output meshes being manifold. Despite this, the visualisations of the output meshes show that the application can produce self-intersections as shown in the close up image in Figure 5.7. The lower right of the image clearly shows a triangle emerging from the mesh which results in a flickering in that region when moved around in the renderer.

Considering the visual results from Blender (Figure 5.6) show very similar looking results to the ones produced by this project (Figure 5.4). Despite this, further examination shows that Blender fails to correct any issues with the mesh, leaving large holes unchanged. However, this also means that unlike the application produced, Blender does not produce extra components on occasion for non-manifold inputs. It should also be noted that Blender produces results in a much faster time frame and is capable of reducing meshes to any degree specified by the user.

5.7 Discussion of the results

Framing the results highlighted by the evaluation, in the context of the aims and requirements of the project is important for determining success and future improvements. The results of the manifold tests highlight that whilst manifold outputs are not always guaranteed due to multiple components being created, they will always be watertight and pinch-point free. Whilst this does not align exactly with the aims of this project, the application is still capable of repairing holes in the mesh to produce hole free surfaces. Further inspection shows that the reason for this issue lies with how the distance fields are produced as the isosurfaces have the issue before the collapse, meaning that NFR 4 has been met. As highlighted in the background research [28], triangle folds and self-intersections can cause issues for signed distance fields. The

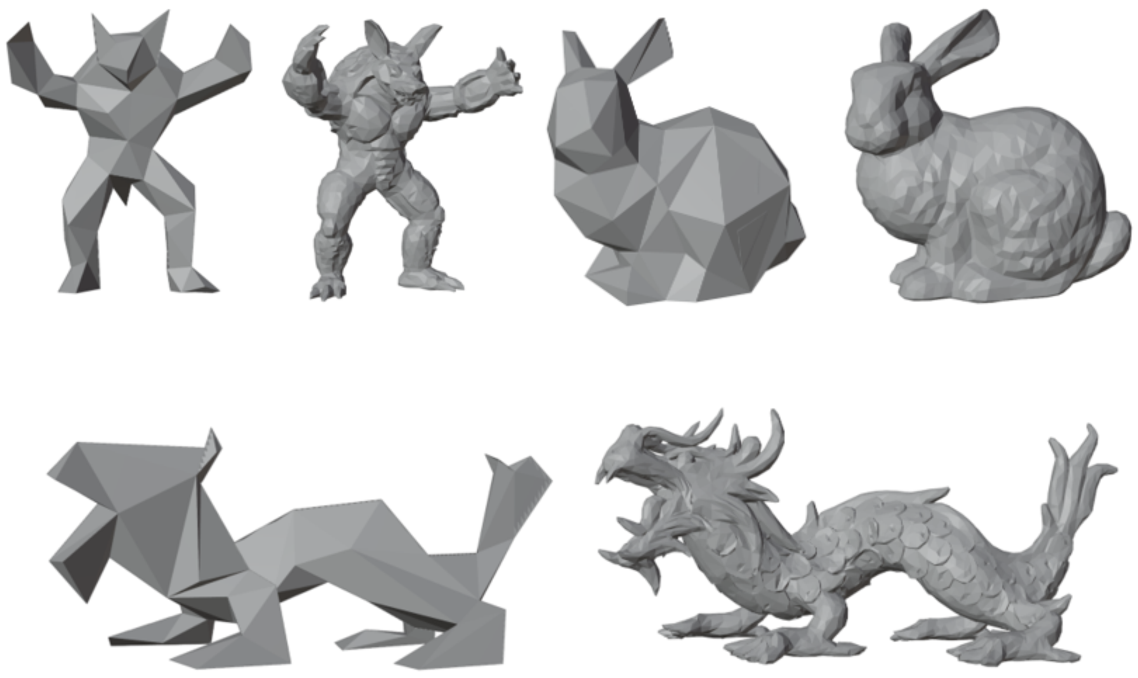


Figure 5.6: Images showing output meshes from the Robust low poly meshing application [8] (left) and Blender [5] (right).

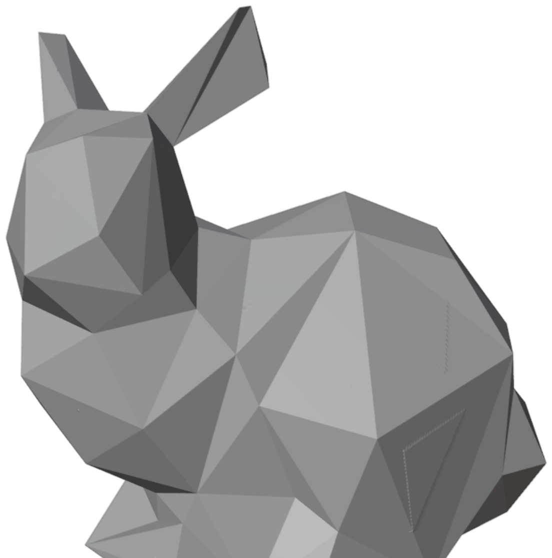


Figure 5.7: Close up of the bunny mesh produced by the Robust Low Poly Meshing application [8].

results seem to confirm this with the extra components only being produced when non-manifold meshes are input. A further explanation could be that the resolution of the distance fields is not high enough causing sections of the mesh to become disjoint. Despite this, 2/3 of the results are completely manifold with 100% of the outputs being watertight as per FR 3.5.

In addition to the manifold tests, the evaluation proves that NFR 1 has been met with the computation time testing. The results show that even with the largest number of triangles, the application runs in just under the time requirement of 2500 triangles per second. On the other hand, the graph in Figure 5.2 shows an exponential growth in execution time for the edge collapse application. This suggests that inputting larger models into the application may result in execution times exceeding the criteria set out. Utilising a larger data set with a more varied polygon count could prove this hypothesis. However, due to time constraints, only 9 input models could be tested with a cap on how large the models could be as to prevent execution times from taking too long. Despite this, the application is successful at meeting the aim for a reasonable execution time however work can still be done in this area. This is especially true when considering the results from the industry comparison (section 5.6) which show sub 1 minute execution times for all models.

The main objective of this project was to generate low-poly meshes that are visually similar to the high-poly inputs as reflected in NFR's 5 and 7. The analysis in sections 5.4 and 5.5 suggest that both of these have been met with the visual comparisons showing barely any difference in shape and the polygon counts being reduced to an average of 3.91% of the original count. This is well below the requirement of 10% outlined in NFR 5 making that requirement very well met. The visual similarity evaluation highlighted that the scale of the model is not preserved which still relates to NFR 7. The reason for this scaling of the model is likely due to the marching cubes algorithm producing the isosurface without knowing the exact distance between grid points in the SDF. This makes all output models conform to a uniform grid of the same scale regardless of the input. Whilst this might not be a desired outcome, it can be seen as a negligible side effect since most models are scaled upon usage anyway. Furthermore, since the model is still proportionate, the scale does not affect the shape of the mesh and does not reduce the overall visual quality.

Commenting on the application in the context of the industrial tools presented by Chen et al. [8] and Blender [5], this project produces comparable outcomes. It is shown that, like this project, the low-poly meshing algorithm [8] can produce topological ambiguities in the output mesh when given non-manifold inputs. Furthermore, although it is not the aim of the application, Blender does not rectify any issues with the mesh. Comparing the outputs visually, all 3 programs result in similar looking meshes with the only difference being the noticeable difference in polygon counts. Chen et al. [8] forces the mesh output to be the lowest possible number of triangles whilst the project aims to allow the user to specify the number of polygons in their output. Likewise, blender does a good job at allowing the user to select any number of triangles for their output mesh whilst doing so in almost real time. The biggest difference between the project's application and the ones used in the industry is the execution time. Blender and the low-poly meshing application are much faster at reducing the meshes, highlighting that for the algorithm to be entirely comparable, significant optimisation must be done.

In summary, the program produced matches the aims of this project very well with the exception of the limitations posed by the distance field method. By running the application, all functional requirements relating to the inputs and outputs of the programs were tested and successful. Furthermore, visual and topological testing highlights that FR 1.3, 2.3, 2.4, and 3.5 as well as NFR 1, 4, 5, and 7 have all been met. Comparing the produced application to industrial examples shows that whilst it is comparable with mesh outputs, the execution time can be massively improved in order to be competitive with the market.

Chapter 6

Conclusions and Future Work

This project aimed to produce an application capable of taking any input mesh and producing a low-poly output mesh for use in video games. By researching the subject area of mesh generation and mesh simplification, a 3 stage process was designed alongside requirements that encapsulated the aims of the project. Using this design, 3 separate applications were developed which, when used sequentially, produce a low-poly version of the input mesh successfully. Finally, the implementation was critically evaluated with the project aims and discussed within the context of the industry.

Given the nature of the project, assessing the outputs produced by the application was key to determining the project's success. Evaluating the outputs visually and topologically highlighted that the main aims of the project have been met with the application generating visually similar low-poly models within a reasonable time frame. Further discussion of the results emphasises that the application does not always produce a manifold output due to the presence of extra components caused by inaccuracies in the distance field computation. Despite this, the outputs produced by the application are comparable topologically and visually to other freely available tools used within the industry. Both tools compared reduce polygon counts to a similar degree with both Blender [5] and the application produced allowing the user to specify the exact number of triangles desired in the output mesh.

Framing the project outcomes in the context of the industry demonstrates areas for improvement, particularly showing that optimising the program is worth exploring in future work. Specifically, it is found that the time taken for the edge collapse algorithm increases exponentially with the number of input polygons. Optimisations could include the parallelisation of the process on GPU or the inclusion of acceleration structures to allow the data to be accessed and modified faster. Furthermore, to guarantee that the output will be manifold, the distance field algorithm should be reworked to fix ambiguities in the mesh. This rework could utilise a more visual approach to prevent the limitations caused by triangle folds and self-intersections.

To summarise, this project has met all aims to a very good extent with the exclusion of a guarantee that the output mesh will be manifold. The report details the steps taken to implement the application along with the techniques used to evaluate the project. Further work from this project should improve the distance field application and optimise the edge collapse algorithm to compete with other tools in the industry.

References

[1] Adobe. The Different Types of 3D File Formats - Adobe — adobe.com. `https://www.adobe.com/products/substance3d/discover/3d-files-formats.html`. [Accessed 29-07-2024].

[2] J. Baerentzen. Robust generation of signed distance fields from triangle meshes. In *Fourth International Workshop on Volume Graphics, 2005.*, pages 167–239, 2005.

[3] J. Baerentzen and H. Aanaes. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):243–253, 2005.

[4] B. G. Baumgart. Winged edge polyhedron representation. Technical report, Stanford, CA, USA, 1972.

[5] Blender Foundation. blender.org - Home of the Blender project - Free and Open 3D Creation Software — blender.org. `https://www.blender.org/`, 1995. [Accessed 27-07-2024].

[6] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Levy. *Polygon Mesh Processing.* A K Peters ; Taylor & Francis distributor, 2011.

[7] H. Carr. (No) More Marching Cubes . In H.-C. Hege, R. Machiraju, T. Moeller, and M. Sramek, editors, *Eurographics/IEEE VGTC Symposium on Volume Graphics.* The Eurographics Association, 2007.

[8] Z. Chen, Z. Pan, K. Wu, E. Vouga, and X. Gao. Robust low-poly meshing for general 3d models. *ACM Trans. Graph.*, 42(4), jul 2023.

[9] E. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical report, 1995.

[10] D. Eberly. Distance between point and triangle in 3d. 1999.

[11] G-Truc Creation. Opengl mathematics (glm). `https://github.com/g-truc/glm`, 2005.

[12] X. Gao, K. Wu, and Z. Pan. Low-poly mesh generation for building models. In *ACM SIGGRAPH 2022 Conference Proceedings*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery.

[13] M. Garland and P. S. Heckbert. *Surface Simplification Using Quadric Error Metrics*. Association for Computing Machinery, New York, NY, USA, 1 edition, 2023.

[14] J. Hasselgren, J. Munkberg, J. Lehtinen, M. Aittala, and S. Laine. Appearance-driven automatic 3d model simplification. In *Eurographics Symposium on Rendering*, 2021.

[15] H. Hoppe. *Progressive Meshes*. Association for Computing Machinery, New York, NY, USA, 1 edition, 2023.

[16] M. Hussain, Y. Okada, and K. Niijima. A fast and memory-efficient method for lod modeling of polygonal models. In *2003 International Conference on Geometric Modeling and Graphics, 2003. Proceedings*, pages 137–142, 2003.

[17] A. Jacobson. common-3d-test-models. `https://github.com/alecjacobson/common-3d-test-models/tree/master?tab=readme-ov-file`, 2023.

[18] M. Jones. 3d distance from a point to a triangle. 01 1995.

[19] M. Jones, J. Baerentzen, and M. Sramek. 3d distance fields: a survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[20] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry*, 13(1):65–90, 1999.

[21] D. Khan, A. Plopski, Y. Fujimoto, M. Kanbara, G. Jabeen, Y. J. Zhang, X. Zhang, and H. Kato. Surface remeshing: A systematic literature review of methods and research directions. *IEEE Transactions on Visualization and Computer Graphics*, 28(3):1680–1713, 2022.

[22] R. Klein, G. Liebich, and W. Strasser. Mesh reduction with error control. In *Proceedings of Seventh Annual IEEE Visualization '96*, pages 311–318, 1996.

[23] L. P. Kobbelt, M. Botsch, U. Schwanecke, and H.-P. Seidel. Feature sensitive surface extraction from volume data. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 57–66, New York, NY, USA, 2001. Association for Computing Machinery.

[24] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, aug 1987.

[25] J. Manson and S. Schaefer. Isosurfaces over simplicial partitions of multiresolution grids. *Comput. Graph. Forum*, 29:377–385, 05 2010.

[26] C. Montani, R. Scateni, and R. Scopigno. A modified look-up table for implicit disambiguation of marching cubes. *The visual computer*, 10:353–355, 1994.

[27] T. S. Newman and H. Yi. A survey of the marching cubes algorithm. *Computers & Graphics*, 30(5):854–879, 2006.

[28] H. Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007.

[29] G. Nielson. On marching cubes. *Visualization and Computer Graphics, IEEE Transactions on*, 9:283– 297, 08 2003.

[30] G. Nielson and B. Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *Proceeding Visualization '91*, pages 83–91, 1991.

[31] OpenMP ARB. Home - OpenMP — openmp.org. `https://www.openmp.org/`, 1997. [Accessed 01-08-2024].

[32] B. Payne and A. Toga. Distance field manipulation of surface models. *IEEE Computer Graphics and Applications*, 12(1):65–71, 1992.

[33] R. A. Potamias, S. Ploumpis, and S. Zafeiriou. Neural mesh simplification. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 18562–18571, 2022.

[34] J. Rossignac and P. Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. L. Kunii, editors, *Modeling in Computer Graphics*, pages 455–465, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[35] S. Schaefer and J. Warren. Dual marching cubes: primal contouring of dual grids. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 70–76, 2004.

[36] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '92, page 65–70, New York, NY, USA, 1992. Association for Computing Machinery.

[37] Simplygon Studios. Simplygon - The Standard in 3D Games Content Optimization — simplygon.com. `https://www.simplygon.com/`, 2006. [Accessed 27-07-2024].

[38] Z. Smith. Thingiverse - digital designs for physical objects, 2008.

[39] L. K. Swen Campagna and H.-P. Seidel. Directed edges—a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–11, 1998.

[40] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder. Removing excess topology from isosurfaces. *ACM Trans. Graph.*, 23(2):190–208, apr 2004.

[41] Q. Zhou and A. Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016.

Appendices

Appendix A

GitHub Repository

The source code for this project is available at:
`https://github.com/Tomizzed2001/LowPolyMeshGeneration`

To compile and execute the code found in the repository follow the README.md instructions.