School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Optimising 3D Models – Mesh Simplification With Quadric Error Metrics

Savan Hathalia

Submitted in accordance with the requirements for the degree of
BSc Computer Science

2023/24

COMP3931 – Individual Project

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Final Report	*PDF file*	*Uploaded to Minerva (10/05/24)*
Link to GitHub repository	*URL*	*Sent to supervisor and assessor. Can also be found on the last page of this report under the Appendix section (10/05/24)*
Link to video demonstration	*URL*	*Sent to supervisor and assessor.* *Link on GitHub page..* *Can also be found on the last page of this report under the Appendix section (10/05/24)*

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

SAVAN HATHALIA

Summary

Many applications require the use of complex, highly detailed 3D models to achieve their desired goals. Typical areas in which they are used include the film industry, gaming, architecture and product design.

These models need to have a sufficiently high quality whilst being able to render in a suitable timeframe and using a relatively low amount of computer resources.

Getting this balance between performance and detail is vital to ensure a successful end product for any party using such models.

This project aims to allow users to import objects into a 3D environment, visualize the polygons it is made of, simplify the object with the desired vertex count, and export this object for their own purposes.

Acknowledgements

Professor Hammish Carr, my supervisor, has played a crucial role in my development of this project and has been a source of inspiration throughout.

Table of Contents

Chapter 1
Introduction and Background Research

1.1 Introduction

3D Models are usually made of meshes, a 3D representation of an object. Meshes are made of polygons which are represented by a collection of faces, edges and vertices that define an object's shape and structure.

Whilst there is no standard polygon count for any given mesh, companies and individuals using 3D models must take careful consideration of this number according to the users and use case of the software they intend to create.

A large polygon count leads to an object with a high level of detail, however processing time and memory usage will increase significantly, possibly damaging performance of the program. This can be a major issue especially in, for example, gaming, where modern games are expected to run at 60 frames per second for a smooth user experience. Too low of a polygon count leads to objects showing signs of visible faceting and can be visually unappealing. This ruins immersion where realism and fidelity are important features of the software.

Mesh simplification is a tool used to reduce the number of polygons of a mesh to accommodate for lack of processing power and to reduce processing time. There are a variety of different techniques to achieve this goal, however, this project will be exploring the Quadric Error Metrics (QEM) method initially proposed by Garland and Heckbert in a University paper (Michael Garland, 1997).

Simplifying complex meshes can result in faster rendering times as less computation and memory is required. A smaller file size is needed as less data is needed to be stored and transmitted. These benefits can improve frame rate and responsiveness of the application, save disk space and bandwidth, and make the model more compatible with different devices.

1.1.1 Aims

The aim of this project is to develop a mesh simplification tool using the popular QEM approach. The primary objective is to implement the QEM simplification and evaluate it against other commonly adopted algorithms. It will outline the details of the QEM method, exploring how it works, why it is an efficient algorithm, and how effective the end results are.

A secondary objective would be to enhance the visualization of the meshes possibly by employing techniques such as normal mapping.

The goal is to provide a user-friendly tool that allows for interactive control over the level of detail of 3D models, which can then be exported and used in other computer graphics applications.

1.1.2 Objectives

1) Algorithm implementation: Implement the QEM mesh simplification technique to achieve efficient reduction of computational complexity of a mesh whilst reasonably preserving geometric details.

2) User Interface Development: Design and develop a user-friendly interface for the mesh simplification tool, allowing users to import 3D models, control simplification parameters and visualize the end result against the original.

3) Visualization enhancement: Attempt to enhance presentation of the models using visualization techniques such as normal mapping giving an aesthetically pleasing representation.

1.1.3 Deliverables

1) A fully functional mesh simplification tool implementing the QEM technique which effectively reduces the complexity of a model whilst preserving the key features associated with it (Linked to Objective 1).

2) Provide a user friendly interface for the mesh simplification tool that allows for the importing of 3D models and interactive control of simplification parameters (Linked to Objective 2).

3) Possibly integrate visualization techniques such as normal mapping into the program providing a visually appealing representation (Linked to Objective 3).

4) Final report that details the background research, methodology and results of the project.

5) GitHub repository containing the source code, documentation, and necessary resources for the tool to run.

1.2 Literature Review

This section will cover the important background knowledge which will be necessary to understand the decisions made later in this project.

1.2.1 Types of 3D Modelling

There are four main types of object modelling; point-cloud, voxel based, boundary representation (BRep) and polygonal mesh. This section summarises each type of object modelling.

Point-cloud 3D modelling technique is typically used for scanning of objects. Instead of using mathematical formulas to combine data into shapes, it uses points along the surface to define the object. The density of points determines the resolution of the point cloud model.

Voxel based form of modelling is made up of 3D cubes which represent the objects. In theory, this form of block modelling can be used to map out any shape or form. This form of modelling is currently used in science to understand volumetric data such as terrain or fluids. It is typically used to represent individual particles that can provide scientists with easy to process data (Spartial Corp, 2022).

BRep modelling is a very common form of modelling in Computer Aided Design (CAD) applications which uses mathematical relationships to represent 3D objects with geometric boundaries between solids and non-solid objects. This allows applications to create a theoretically “perfect” model which can be used by designers and engineers to create the CAD model.

Polygon meshes are a computer graphics technique for creating 3D objects which comprises of vertices, edges, and faces that define the shape of the object. It is used often in the gaming industry, animation industry, virtual reality, and CAD. The main reasons that polygon meshes are used is because modern computers find it easier to handle polygons for rending and visualisation. Build-up of polygons means that designers can create unique and realistic looking objects such as animals or humans. Using the smaller polygons allows easier manipulation of each surface creating a more natural movement of the object. The downside is that the more control there is of the mesh (with smaller polygons), the more time it takes for the application to process the data and hence, a balance must be achieved.

Each type of modelling has its own use case and users will want to choose what the best modelling technique is for their requirements.

For this project, the target use case for the application would be in the gaming industry to render objects using polygonal meshes. The scope of this project has been defined to ensure that the project can be realistically achieved in the given timeframe of the course.

1.2.2 Convex and Concave Polygons

Polygonal meshes consist of vertices, edges and faces which define the shape of the object. Faces usually consist of simple convex polygons (e.g. triangles, quadrilaterals, pentagonal etc.). A convex polygon is defined to be when a line between two points of the polygon is contained within the interior of the polygon and its boundary. A concave polygon has at least one internal angle that is larger than 180 degrees. Convex and concave polygons are visualized in Figure 1.

Convex shapes take precedence over concave ones due to them being simpler to work with when implementing algorithms such as collision detection. There exist simple algorithms which can quickly determine whether two convex shapes collide, including separating the axis which states that if you are able to draw a line to separate two polygons, then they do not collide (Chong, 2012). This makes convex shapes far more popular and compatible with other graphics applications.

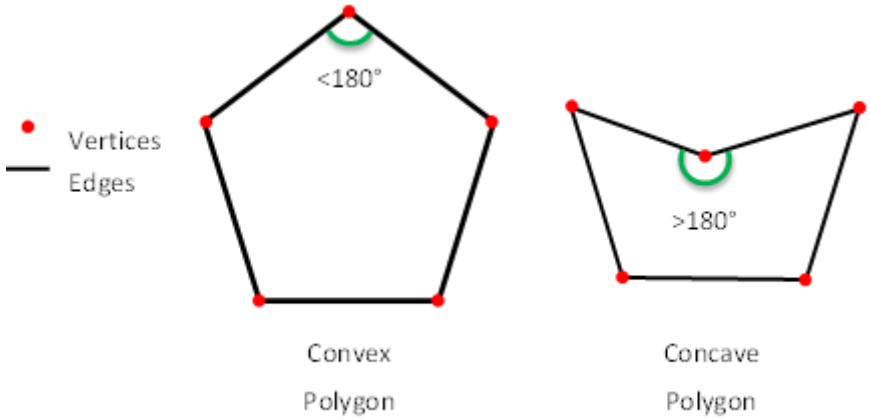


Figure 1: Convex and Concave Polygons

1.2.3 Planar and Non-planar Polygons

When an individual polygon in a mesh is completely flat, it is called planar. All the vertices of that polygon lie on the same plane (see Figure 2).

Triangles are always planar since any 3 distinct points define a plane. Every other polygon can be non-planar, its points might not all lie on the same plane, which therefore means it cannot be rendered. Many applications such as OpenGL use what's known as tessellation when polygons with more than 3 vertices are present.

Tessellation is when a set of polygons is divided into suitable structures for rendering. In OpenGL's case, those polygons would be decimated into triangles. This pre-computation does not need to take to place if the mesh is already made up of only triangles, also making it compatible to graphics applications which do not support tessellation.

For these reasons and since they are most common type of polygonal mesh, triangular meshes are the main focus of this project.

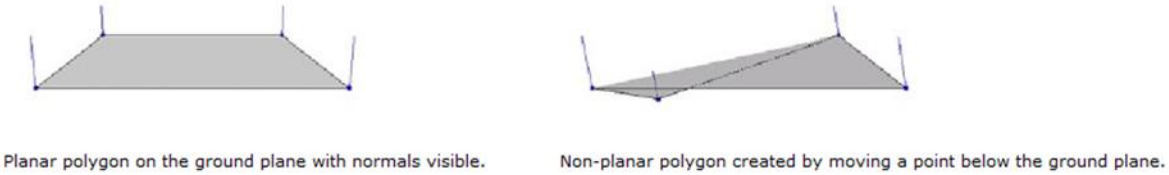


Figure 2: Planar and Non-planar Example

1.2.4 Types of Mesh Representation

Polygonal meshes may be represented in a variety of ways, using different methods to store the vertex, edge and face data.

Listed below are some of the ways they can be represented:

- Face-Vertex Meshes – A simple list of vertices, and a set of polygons (usually triangles) that point to the vertices used.

- Winged-Edge Model (see Figure 3) – Where each edge points to 2 vertices, 2 faces and 4 edges (clockwise abbreviated as CW and counter-clockwise as CCW) that touch them (Bruce, 1972).

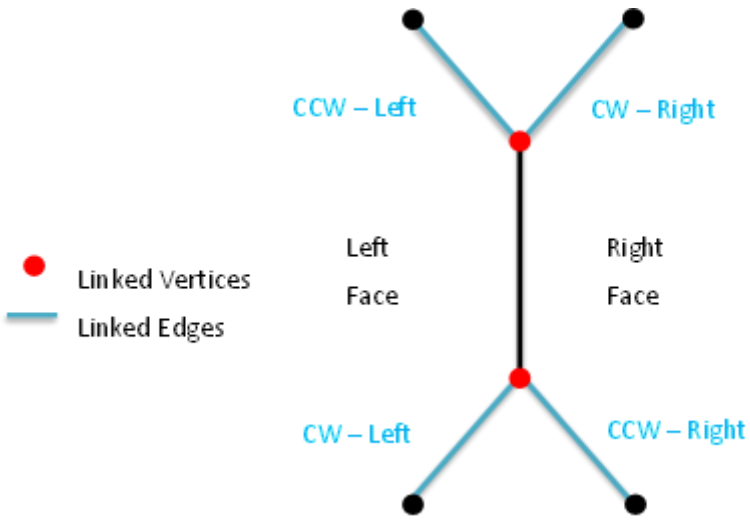


Figure 3: Winged-Edge Representation Model

- Half-Edge Meshes (see Figure 4) – Similar to winged-edge meshes except that only half the edge traversal information is used. A half-edge stores a reference to its twin half-edge, next half-edge, origin vertex and the incident face (Tobler, 2006).

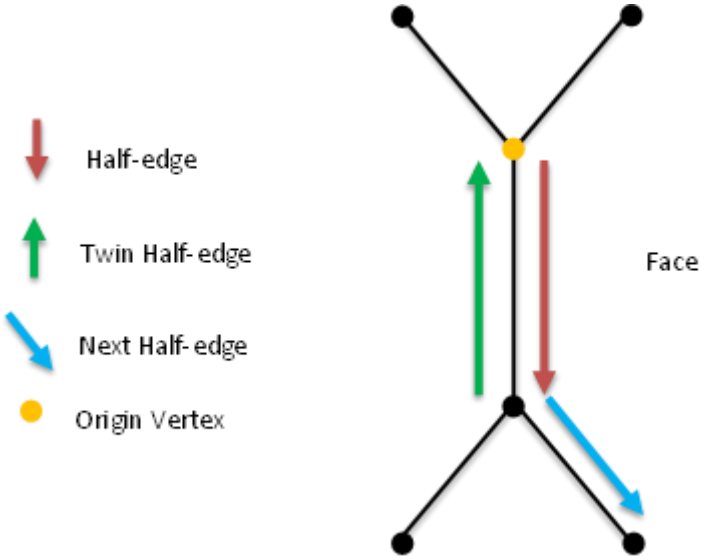
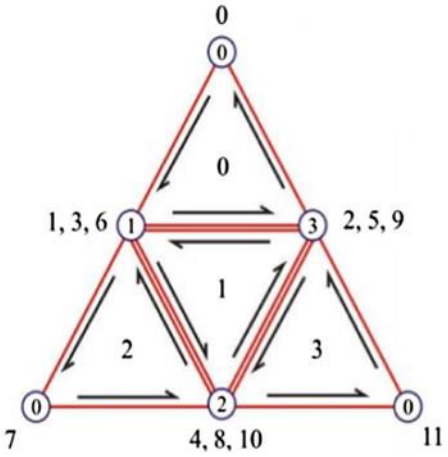


Figure 4: Half-Edge Representation Model

- Corner Tables – Corner tables store vertices in a predefined table, such that traversing the table implicitly defines polygons. The representation is more compact, and efficient polygons, but operations to change polygons are slow. Furthermore, corner-tables do not represent meshes completely. Multiple corner-tables are needed to represent most meshes. Figure 5 shows an example of a tetrahedron and its elements using the corner table structure.



Corner	Vertex	Triangle	Opposite
0	0	0	2
1	1	0	2
2	3	0	2
3	1	1	0
4	2	1	0
5	3	1	0
6	1	2	3
7	0	2	3
8	2	2	3
9	3	3	1
10	2	3	1
11	0	3	1

Figure 5: Corner Table (Bíscaro, 2016)

This list is not exhaustive but are some of the more common ways they are represented. Choosing which method to use when implementing the QEM will be an important factor in how it will be structured and how efficient it can be.

1.2.5 Mesh File Formats

There exists a plethora of file formats for which polygonal mesh data is stored. Popular formats include .fbx, .dae, .obj and .stl. Some programs even create their own formats including the .blend format used by Blender, a prominent 3D computer graphics software.

The file formats that my application will support will need to be considered when it comes to importing and exporting meshes.

1.2.6 Existing Solutions

Almost all 3D graphics software now come as standard with built-in mesh simplification tools. The techniques each of these use can vary substantially depending on the type of application, the way it represents meshes, and its use cases.

An open-source application called MeshLab, which describes itself as a system for processing and editing 3D triangular meshes, includes simplification algorithms such as the QEM within its editor.

In the gaming industry, there exists a commercial software designed specifically for automatic and manual LOD (level of detail) generation and mesh simplification called Simplygon. It uses proprietary methods so details of their methods are not shared publicly. However, they have been licensed by a number of AAA game studios for use in prominent titles including God Of War and Spider-Man 2.

Blender, a free, open-source 3D computer graphics software tool used for modelling, animation, visual effects, art and much more, contain various techniques for simplification of models. These include “collapse”, “unsubdivide”, “planar”, “limited dissolve” and “optimised”. Users can achieve desired results tailored to their specific needs by selecting the appropriate technique and adjusting their associated parameters.

The next chapter will include details on the different simplification methods that will clarify exactly how the aforementioned methods work and the need for each of them.

Chapter 2
Methods

2.1 Design Choices and Justifications

2.1.1 Programming language – C++

C++ was chosen as the programming language for this project for a number of reasons:

- Performance: C++ has a high performance due to it being a compiled language, code is converted into machine code fully before it is executed. Compared to interpreted languages such as Python, where one line is executed at a time, compiled languages tend to run faster. This is ideal for resource intensive applications such as graphics applications
- Libraries: C++ has a vast array of libraries and frameworks tailored for graphics programming including OpenGL, OpenMesh and ImGui, all libraries that will be used in this project. These libraries provide powerful tools and functionalities that will aid with the project
- Integration: Many existing graphics and game engines are written in C++ (e.g. Unreal Engine and Unity). Using C++ means this project can be easily ported to these applications, enabling leverage of their features and workflows
- Learning Experience: Working with such a low-level programming language helps deepen my personal understanding of programming concepts, memory management, and performance optimisation
- Familiarity: C++ is one of the many languages I have experience in writing. Other options were Python or Java. Although, because of the other reasons listed, C++ seemed to be the strongest option
- Popularity: According to a survey conducted by Stackscale in 2023, C++ was ranked 5th in the most popular languages. Almost all other research gathered from other vendors had C++ in the top 10 at least

2.1.2 Platform – Windows

Windows was the preferred platform to support first because of:

- User Base: Windows by far has the largest user base worldwide when it comes to operating systems. A survey done by Statista shows around 70% of people use Windows. From the questionnaire I have created, it also indicates all stakeholders

that filled it out use Windows. Developing primarily for this platform ensures a board reach and accessibility in the market

- Development Tools: It offers a range of development tools including the integrated development environment (IDE) Visual Studio, which provide comprehensive features for C++ development and debugging
- Compatibility: Many devices and third-part software applications are designed with Windows compatibility in mind
- Gaming Market: It remains the dominant platform for PC gaming according to the Steam charts which indicate 96% of their user base uses Windows. Developing the tool for this OS means it can easily be used in games with Windows as their main target audience

2.1.3 Version Control – GitHub

GitHub was used since it offers several benefits for software development including:

- Version Control: GitHub provides powerful, easy to use version control features, allowing developers to track changes to the codebase and manage project history quickly. Reverting changes becomes a simple task, making the workflow much more efficient
- Project Management: GitHub offers useful project management features such as project boards and task tracking allowing organisation of tasks

2.1.4 IDE – Visual Studio

Visual Studio (VS) was by far the best choice when it came to choosing the IDE:

- Rich Feature Set: VS offers a comprehensive set of features and tools for software development, including code editing, debugging, testing, version control integration and more
- Clean User Interface: It provides a customisable user interface where developers can easily navigate between different views, windows, files and tools.
- Language Support: Many programming languages are supported by VS including C++
- Conveniences: Linking libraries and including external directories is a simple procedure in VS. Files that are not C++ files, such as shader files can still be created and changed within its editor

2.1.5 Graphics API – OpenGL

A graphics application programming interface (API) is a type of API that creates a bridge of communication between a particular software application and the integrated or dedicated graphics processing unit within a computer.

There were 3 main Graphics APIs which were considered for this project; DirectX, Vulkan and OpenGL. These are all graphics APIs used for rendering 2D and 3D graphics in applications such as games, simulations, and visualizations. They have similar purposes but significant differences in the architectures, features, performance, and target platforms of the APIs. Here is a brief comparison of DirectX, Vulkan and OpenGL:

- Architecture:
  - DirectX: A set of APIs developed by Microsoft which include the Direct3D API, Direct2D for 2D graphics, and many more.
  - Vulkan – A low overhead, explicit, cross-platform API developed by the Khronos group that provides great control over hardware resources and parallelism. This control allows for more efficient utilization of modern GPU architectures although more work has to be done on the developers side.
  - OpenGL: OpenGL (Open Graphics Library) is an older, high-level graphics API also developed by the Khronos group that provide a set of functions to render graphics primitives and control rendering state. Due to the existence these higher level functions, OpenGL is a simple API to learn.
- Performance:
  - DirectX – Latest versions of DirectX offers high performance and low-level access to hardware on Windows platforms
  - Vulkan – Generally gives better performance than OpenGL, and in some case DirectX due to the explicit control it allows over GPU resources and efficiency of command submission. Unlike OpenGL or DirectX pre-11, GPU commands have to go through a command buffer and executed through a queue.
  - OpenGL – OpenGL usually has higher CPU overhead and a less efficient multithreading model compared to Vulkan. However, this has been optimised throughout the years so generally gives decent performance for most applications.
- Platform Support:
  - DirectX – Aimed directly at Windows platforms and does not natively support any other.
  - Vulkan – Supported on multiple operating systems, including Windows, Linux, Android and even some MacOS platforms. Offers equivalent performance and behaviour across devices and platforms.

- OpenGL – Like Vulkan, this is also cross platform and has wide support for almost every OS including Windows, Linux, MacOS, Android and IOS.

OpenGL was eventually decided as the best options since it was one that I had some experience in, whilst also being available on Windows and the potential to be cross-platform. Performance wise, since my application will not be too resource intensive, OpenGL will run to a good standard and the extra performance of Vulkan or DirectX won't be essential.

2.1.6 User Interface – ImGUI

When choosing which user interface API to use, ImGui and Qt were the 2 options that were considered. They are modern GUI libraries for developing user interfaces. Whilst both have similar purposes, they adopt different design philosophies, features and use cases. Let's compare:

- Purpose
  - ImGui – A lightweight GUI library predominantly used for use in game development and tools/applications where simplicity and performance are top priorities. It has an easy-to-use API for UI creation, allowing for this creation right in the renderer loop. This makes the library easy to embed with most graphics frameworks
  - Qt – A full featured cross-platform framework that provides a vast number of tools and libraries for GUI and non-GUI applications. It supports development with desktops, mobiles, and embedded systems with feature support for widgets, layouts, graphics and more
- Design Philosophy
  - ImGui – Follows the "immediate-mode" GUI paradigm, where UI elements are created and rendered each frame. Allows for creation of custom UI components quickly and efficiently and focuses on simplicity, ease of use with minimal overhead
  - Qt – Follows the "retained-mode" paradigm, where UI elements are created and manged as objects in a scene graph. It comes with a number of pre-built widgets and layouts, with features for event handling, styling and theming
- Features
  - ImGui – Includes windows, buttons, text inputs, sliders and image display as some of its core functions and GUI primitives. It has a simple API to be able to customize styles and layouts, and allows for customs widgets and extensions
  - Qt – Provides a full set of GUI tools, from standard widgets like buttons, labels, and text editors, to advanced concepts such as animations, gestures, and touch input

- Integration
  - ImGui – Easy integration with graphics frameworks including DirectX, Vulkan and OpenGL. It does not depend on external libraries and therefore can implemented into the codebase directly
  - Qt – Provides its own IDE (QtCreator) and requires more understanding of specific methods Qt uses to integrate with graphics APIs
- Platform Support
  - ImGui – Platform independent, runs on various platforms including Windows, Linux, MacOS, Android and IOS
  - Qt – Cross-platform framework again able to run on Windows, Linux, MacOS, Android and IOS. However, Qt provides a native look and feel on whichever OS it is used on, guaranteeing high-quality finishes on different devices

ImGui was decided to be the UI library used for my case because of it being lightweight and easy to integrate into OpenGL compared to Qt. It provided all the features that I would need to use and the extra tools and functionalities offered by Qt were not needed.

2.1.7 Type of Models – Triangular Polygonal Meshes

Triangular polygonal meshes will be primarily supported by my application since these are the most commonly used in the graphics department. Several advantages are present with triangular polygonal meshes over other types such as quadrilateral or n-gonal meshes. These include:

- Simplicity: Triangular meshes are straightforward to work with since triangles are convex and algorithm implementations are simpler in many graphics related tasks such as rendering, collision detection and mesh processing
- Consistency: Representation is consistent leading to easy understanding of topology and connectivity of the mesh. With triangles, every face has the same number of vertices, edges, and faces linked to it
- Rendering Efficiency: Most rendering algorithms and architectures are optimized for triangles leading to faster rendering times
- Planarity: Since triangles are all planar, there will not be a need for pre-processing techniques such as tessellation, which makes rendering speeds slightly faster

2.1.8 File Format – .obj

Supporting the .obj file format was a logical option due to a few factors:

- Widely Supported: One of the most common formats for exchanging 3D models between applications. Proposed by a former company called Wavefront Technologies in 1992 and is still in use, currently supported by a broad range of 3D modelling and rendering software including Blender, Maya, 3DS Max and many others
- Simplicity: Relatively simple to read and understand which makes parsing an easier task. Uses plain text files with a simple structure, making it easy to create and manipulate them
- Features: Supports essential features for 3D models such as vertex positions, normals, texture coordinates, and face indices
- Learning Experience: For developers that are new to 3D graphics such as myself, implementing support for the .obj format is a great way to understand geometric data structures, efficiently loading and rendering models and parsing text files

2.1.8.1 Basic .obj structure

The structure of .obj files include:

- List of geometric vertices, with (x, y, z, [w]) coordinates, “w” is optional and defaults to 1.0. “w” is known as the homogenous vertex coordinate. It is a factor which divides the other vector components. When w is 1.0, the homogenous vertex coordinates are “normalized”. Apart from scaling vertex coordinates, the w coordinate is required because of the existence of 4x4 matrices (model, view and/or projection) and 4x1 matrices (the vertex).
  Example: “v 0.123 0.234 0.345 1.0”, where “v” stand for the vertex position
- List of texture coordinates, in (u, [v, w]) coordinates, ranging from 0 and 1. “v” and “w” are optional and default to 0. “u” and “v” refer to the texture coordinate of a 2D textures similar to “x” and “y” coordinates.
  Example: “vt 0.5 1 [0]”, where “vt” denotes a texture coordinate
- List of vertex normals in (x, y, z) form, usually these are unit vectors. Unit vectors are where the coordinates of the normal are divided by its length to obtain its direction vector relative to each unit of the coordinate system.
  Example: “vn 0.707 0.000 0.707”, where “vn” denotes a vertex normal
- Polygonal face elements in the form (v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3) where each face is made up of 3 vertex position indices and each vertex position is linked with a texture coordinate and normal index. Note that the face does not need to include texture or normal coordinates and the form it would be represented with has slight variation.
  Example: “f 6/4/1 3/5/3 7/6/5”, where “f” denotes a face element

An obj file can reference a separate .mtl file for material properties such as colour, shininess and transparency. Support for materials like this are beyond the scope of this project so will not be explored in detail.

2.1.9 Meshes

For the selection of meshes to be tested in this application, the Stanford 3D Scanning Repository was used. This repository contains detailed reconstructions of scanned models created by various individuals in Stanford University. These have been made publicly available for use as long as credit is given to the Stanford Computer Graphics Laboratory and not used for commercial purposes.

Models in this repository are all made of triangular polygonal meshes which is the target model type my application should support.

Unfortunately, the file format given on the official repository website are .ply files. A Github repository was found that contained these models converted into .obj files, which is the format to be supported initially.

Figure 6 shows an example, the “Stanford Bunny” model that will be used to test my application. It contains around 35,000 vertices and 69,000 faces. The number of faces is significantly larger than the number of vertices since multiple faces can link to the same vertex.

Figure 6: Stanford Bunny Model

Models from this repository have been used extensively in scholarly articles for various purposes, from simplification to feature detection to forming patterns and much more. This is

due to the source being a credible one, the models being made publicly available, and the great detail of each model consisting of a substantial number of vertices.

2.1.10 Simplification Method – QEM

There are 3 main classifications of simplification methods; vertex clustering algorithms, incremental decimation, resampling algorithms. A large portion of information from the book Polygon Mesh Processing by Mario Botch (Mario Botch, 2010) was used for this section.

2.1.10.1 Vertex Clustering Algorithms

Usually very efficient and robust. Computational complexity is typically linear to the number of vertices of the mesh. Quality may not be satisfactory compared to other methods.

It works as follows. For a given approximation tolerance, we partition the space around the object into cells with diameter smaller than that tolerance. For each cell, compute the representative vertex position, which is assigned to all vertices within that cell, all vertices in that cell are clustered into 1. Remove all degenerate faces and you obtain the simplified mesh.

One disadvantage of this approach is that the resulting mesh may no longer be 2-manifold. 2-manifold meshes are ones where each edge is shared by at most 2 faces and each vertex is shared by 1 or more face. This can lead to non-planar faces, which makes texture mapping or lighting more difficult since these operations assume planarity. Removing the 2-manifold property can lead to issues when performing geometric algorithms such as collision detection become an issue.

2.1.10.2 Incremental Algorithms

In most cases lead to higher quality meshes. Complexity in the average case is O(nlogn) but can go up to O(n2) where n is the number of vertices.

It involves removing 1 vertex at a time, where in each step, the candidate for removal is determined based on user specified criteria. This criteria can be binary (removal is/isn't allowed) or continuous (rate the quality of the mesh after removal).

Binary criteria can refer to the global approximation tolerance or minimum requirements such as the minimum aspect ratio of triangles. Continuous criteria can include the comparison of isotropic triangles versus anisotropic ones, or the distance of normal jumps between neighbouring triangles.

The 3 main incremental decimation techniques include vertex removal, edge collapse and the half-edge collapse.

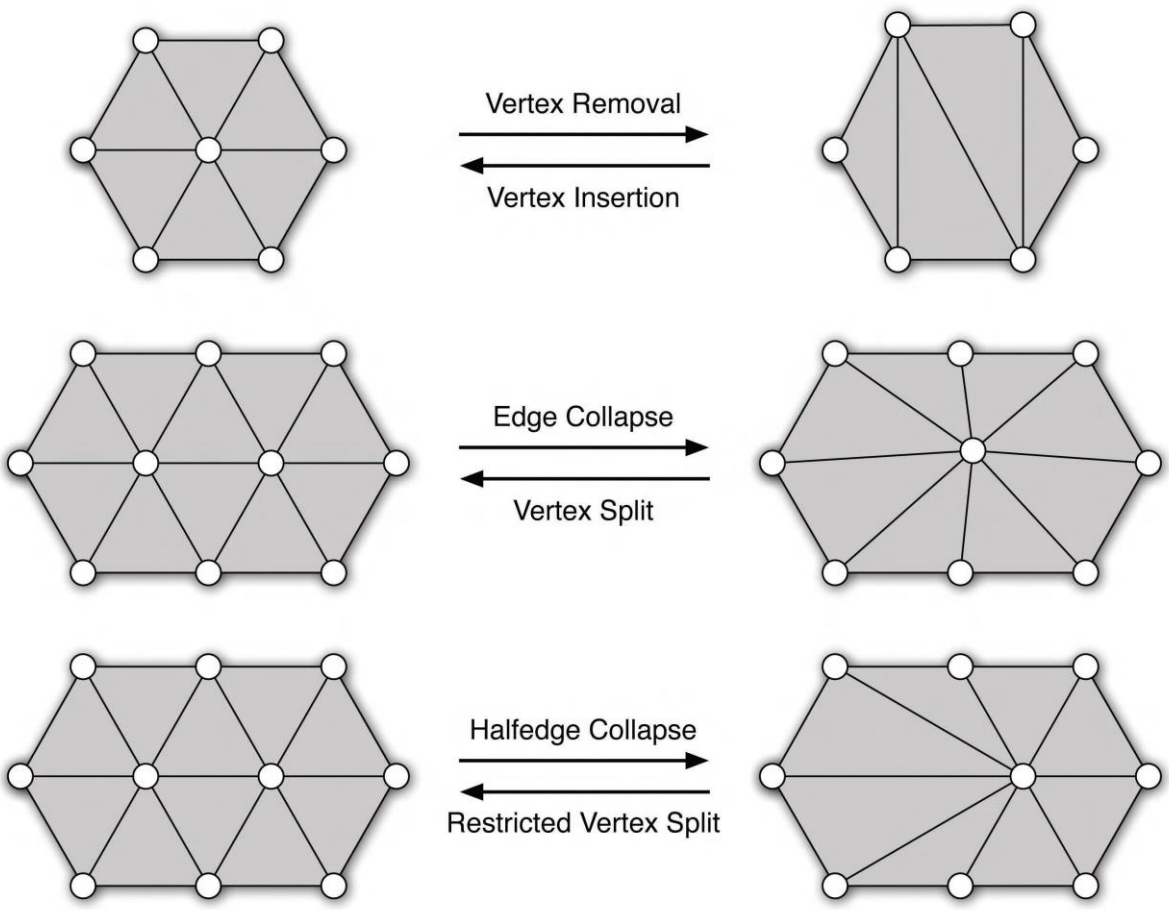


Figure 7: Incremental Decimation Techniques

- Vertex removal involves deleting a vertex, deleting its edges, then reconnecting the neighbouring vertices. For a vertex with degree (number of edges connected to it) *k,* a *k*-sided hole will emerge and reconnecting these vertices will always lead to *k*-2 triangles. It therefore removes 1 vertex, 3 edges and 2 triangles (see Figure 7)
- Edge collapse takes 2 adjacent vertices, and collapses the edge between them so both vertices are moved to the same new point (see Figure 7). This operation removes 1 vertex, 3 edges and 2 triangles
- Half-edge collapse also takes 2 adjacent vertices, and collapses the edge between them, however the resulting point ends up being one of the original points (see Figure 7). This means it can be considered either a special case of edge collapse, or a special case of vertex removal

Decimation by itself isn't the most effective approach and can lead to unsatisfactory results. This is where calculating error before each iteration of decimation comes into play. The main techniques to obtain some error metric includes error accumulation and error quadrics.

- Error accumulation initially proposed by Schroeder (William J. Schroeder, 1992) is a simple technique. An example is with edge collapse, the adjacent triangles have their

vertices shifted, and this shifted distance is then added to the stored error value of each of those triangles. This can not only be done with scalar distances, but also vectors, in which case, the approximation error takes into account if the vertex has been moved in the opposite direction again

- Error quadrics, where each vertex is assigned a distance value (a quadric) which is based on estimating the sum of squared distances from that vertex to all of its planes (all triangles it connects to). The quadrics associated with the 2 vertices of an edge in the mesh are then used when calculating the error of collapsing it to the new point (Michael Garland, 1997). This is in fact the technique proposed by Garland and Heckbert and the algorithm I will attempt to bake into this project

Blender's "collapse" and "planar" functions both use incremental decimation. The "planar" option specifically looks for more than 2 triangles that have the same plane. These can be can be reduced down to only 2 triangles with no impact on the model quality.

2.1.10.3 Resampling Algorithms

Here, new sample points are taken across the surface of the mesh. By connecting these samples, the simplified mesh is formed. They have a special property in that they can be used to create multiresolution representations (essentially different LODs) based on one subdivision basis function (M. Eck, 1995). A disadvantage of this method however, is aliasing, the appearance of jagged edges in a rasterized image. Aliasing can happen when the sampling pattern does not directly align with the features of the original geometry.

Simplygon's methods may involve resampling algorithms due to the ease of LOD creation.

Incremental decimation using error quadrics was chosen for this project since it was easy to understand compared to resampling algorithms, whilst also providing high quality end results unlike vertex clustering algorithms.

2.1.11 Libraries and Dependencies

A number of open-source libraries will be used to aid with this project. Here is a description of each and what they will be used for:

- OpenGL – Allows interaction with the GPU which is instrumental in rendering and visualizing the 3D models. The justifications of using OpenGL were mentioned in section 2.1.5
- GLFW – Lightweight library for creating windows and managing OpenGL input events, which will be needed for to move around the scene with user inputs

- GLAD – A library for loading OpenGL function pointers at runtime, allowing access to OpenGL functions from any platform. This makes the application platform independent since it loads OpenGL functions it needs dynamically
- ImGui - User interface library. Justifications were mentioned in section 2.1.6
- Dirent.h – Header file in C++ that provides functions for directory handling which can be used to iterate through contents of a directory. Will be required for loading in .obj files from the user's file system
- ImGuiFileDialog – A directory handling tool with the ImGui interface. Instead of only using dirent.h which would open the windows file explorer, with ImGuiFileDialog, the directory will pop up in an ImGui menu. This makes the user interface more consistent and intuitive to use
- GLM – GLM is a header-only C++ maths library for graphics programming, providing classes and functions to perform vector and matrix operations. This will make many common math operations easier to implement such as transformations, projections, and operations involving vectors and matrices
- OpenMesh – Open-source library for storing, manipulating, and processing polygonal meshes that supports a range of data structures and algorithms. It also includes the quadric error metrics simplification within its files, which will be used as a way to compare my algorithms results with OpenMesh's results. It is a fallback in-case for if I am unable to implement my own version of the quadric error metrics
- Assimp – Enables import of various 3D model formats into a single, common data structure, allowing access and manipulation of models from a variety of sources. It supports many file formats including .objs, .plys (which is the format the official Standford repository provided), .fbx and many more. This allows not only the initial .objs to be supported for my application, but in future can expand to allow other types. I will be implementing my own extraction of .objs, although this is another fallback I can use in-case things don't go as planned

2.2 Quadric Error Metrics

This section will look further at the details of the Quadric Error Metrics technique including the mathematics required to implement it.

Finding the error of a vertex requires calculation of the quadrics of each vertex **v**. The error is essentially the sum of squared distances from **v** to all planes it connects to. In mathematical notation it would look as follows:

$$\sum_{p \in planes(v)} (p^T v)^2$$

where **p** = [a, b, c, d]^T is a plane in the set of planes associated with **v** and represented by the plane equation **ax + by + cz + d = 0**. T represents transposing the vector. Rewriting this equation gives

$$\sum_{p \in planes(v)} (v^T p)(p^T v)$$
$$= \sum_{p \in planes(v)} v^T (pp^T) v$$
$$= v^T \left(\sum_{p \in planes(v)} p^T p \right) v$$

Let

$$Q = \sum_{p \in planes(v)} p^T p$$

Q is the fundamental error quadric that can be used to find the error of the new position of a vertex. The new vertex position $\mathbf{v_n}$ in this case will be the midpoint of the edge between the 2 vertices $\mathbf{v_1}$ and $\mathbf{v_2}$ associated with that edge.

For every edge in the mesh, the error of the new vertex is then calculated as follows:

$$error = v_n^T (Q_1 + Q_2) v_n$$

where $\mathbf{Q_1}$ refers to the fundamental quadric of $\mathbf{v_1}$ and $\mathbf{Q_2}$ refers to the fundamental quadric of $\mathbf{v_2}$.

It is then a case of collapsing the edge with the least error value to the new position $\mathbf{v_n}$, making sure to remove the degenerate faces, and reconnecting the vertices neighbouring $\mathbf{v_1}$ and $\mathbf{v_2}$.

This collapsing removes 1 vertex at a time, so this will continue to loop, calculating the new errors associated with the vertices whose connections changed and the new vertex, until the desired vertex count is reached.

2.3 Software Architecture

2.3.1 Object-Oriented Approach

Using object-oriented programming (OOP) for this project proved the best option since it offers several benefits.

Modularity and Encapsulation - Allows clustering of related data and logic into objects, making it easier to manage and maintain the codebase. Parts of my applications such as meshes include well-defined characteristics such as vertices, edges and faces, which can easily be encapsulated into a mesh object.

Code Reuse - Objects created from a class includes all features and can use any of its functions without having to rewrite code specifically for that object.

Abstraction - Makes the code easier to read, making decisions about how to move forward much more efficiently.

Overall, using OOP provides a structured and organized approach to developing a complex program such as this mesh simplification tool with many moving parts, helping manage complexity and promoting code reuse.

2.3.2 UML Class Diagram

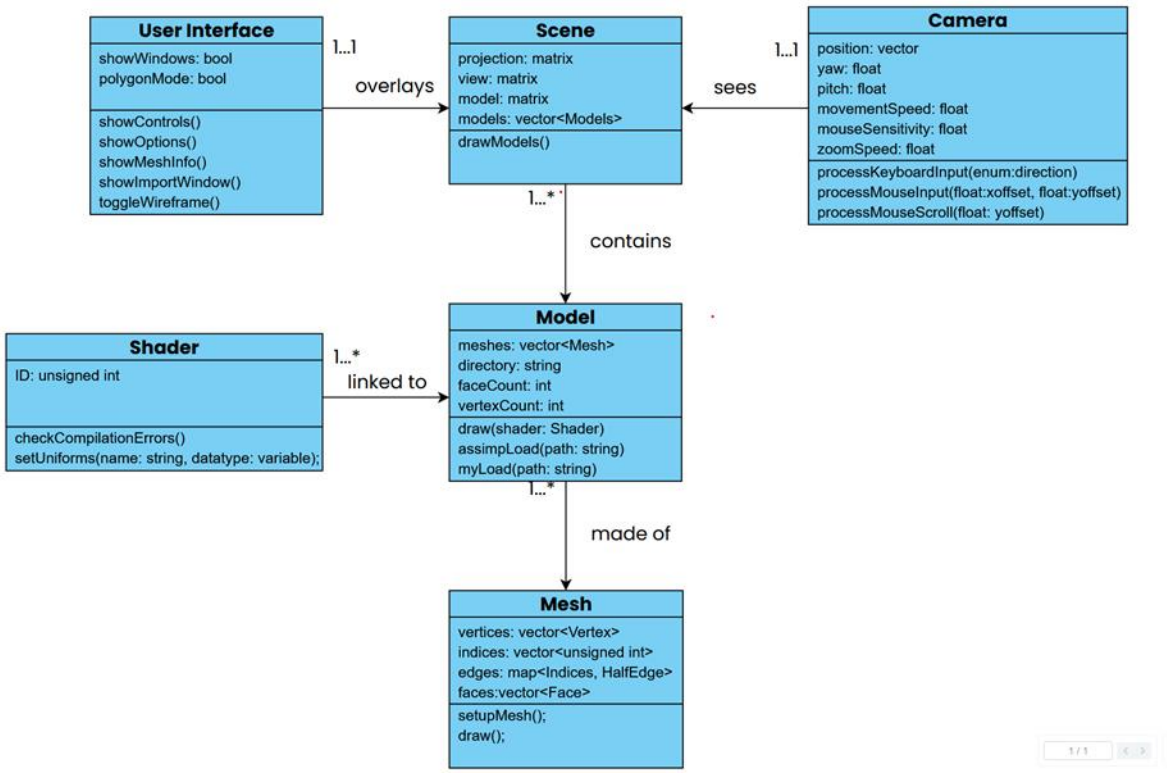


Figure 8: UML Class Diagram

A broad overview of the class structure is shown in Figure 8. It consists of a 3D scene/world which can contain many models. Every model (extracted from the .obj file in my case) can be made up of multiple meshes. A shader is linked to each model in the scene which dictates its lighting behaviour. The user interface will overlay the scene, and there will be a camera from which the user views the world from. Below is a brief summary of what each class will include.

User Interface: Functions to show the different windows including controls, mesh info window and importing window. Users will be able to toggle “wireframe mode” which will change the model visualization to show the triangles it is made of

Scene: Relevant matrices required to calculate the position of all objects in the 3D scene including lighting objects and the models. It will handle initialisation of certain variables such as the creation of model objects

Camera: Parameters required for camera logic including camera speeds on input, rotation angles on input, and storing the position of the camera, which is essential when drawing to the screen

Model: Storing a vector/list of meshes that it consists of, the directory of the .obj file and attributes such as number of faces and vertices. Functions include loading the model and handling the drawing of the model.

Mesh: Each mesh holds vertex information, the indices (linked to a vertex) that make up the faces of mesh, the edges which will map a pair of indices and link them to a half edge structure, and faces.

Shader: Will handle any shader compilation errors and handle the relevant parameters associated with the lighting of a particular model.

This is a general summary of the key components of this software. In practise, there will be many more variables and functions required for it to run as intended, and these will be discussed further in the results and discussion section.

2.4 Project Management Methodology

Sprints, an agile methodology, was the chosen approach when it came to developing this project. Justifications of using this method include:

- Iterative Development: Allows manageable iterations for my project with focus on small, incremental improvements to the software. This gives the possibility to have continuous feedback to adapt to changes in conditions of the project.
- Flexibility: Changing requirements and priorities becomes an easier task. This is necessary for complex software such as a 3D simplification tool, where requirements can change as the problem domain reveals itself or unexpected challenges pop up.
- Risk Mitigation: At the end of each sprint, problems or risks can be identified and fixed. Issues are found earlier and reduces problems later in the project.

For this project, each sprint will be variable depending on the depth of each task, some being 2 weeks, others being 3-4 weeks.

A Gantt chart was used to keep track of which components were being worked on, how far they are to completion, and how long each task took. In the Appendix section, you will find the Gantt chart created for this project.

Chapter 3
Results and Discussion

3.1 Implementation

The first step towards implementing this solution was learning OpenGL. Although I had some knowledge of this API, it was not enough to create such a complex project. Learnopengl.com was used as the primary resource for this task which is a website based on the book by Joey de Vries (Vries, 2020). It contained information ranging from the basics of OpenGL such as creating a window and setting up a 3D environment, to advanced topics such as physically based rendering. Parts I – Getting started, II – Lighting and III – Model Loading were the 3 parts of this book used for this project, the rest were beyond the scope.

3.1.1 Getting Started

- Downloading and linking the correct libraries as well as importing the relevant source code files was mostly done first, setting up the foundation to work from. Theses libraries included the assimp, glad, glfw, glm and openmesh libraries. The external source code included imgui related files and dirent.h. Descriptions of what these libraries do were explained in section 2.1.11.
  Test: To test the linking and importing processes were done correctly, functions from each library were called. If there were any compilation issues, it most likely would be due to incorrect linking
- Setting up a window where the scene would be viewed.
  Test: Run the program and see if a window popped up
- Drawing a basic triangle to the screen was done to refresh knowledge of buffers used by OpenGL.
  Test: See if the triangle was drawn correctly to the screen
- Majority of the shader logic was implemented next, including error messages if compilation was unsuccessful, and helper functions to set variables/uniforms that would be passed into the shader.
  Test: Purposely make syntax errors in the shader files to see if the error messages were shown correctly. Test basic shapes such as the triangle linked to the shader to see if it works as intended
- Although basic implementation for textures were included, these will not be used since it was too complicated to handle simplified meshes' textures.
  Test: Check if textures were added to the triangle

- Camera and input were setup next, including all the required matrices and transformations for the logic of movement and projection.
  Test: Input all controls that were implemented to check if the desired result were achieved i.e. the correct transformations of the world and camera were correct

3.1.2 Lighting

- Phong lighting implemented in the shaders which involves 3 attributes of an object, its ambient lighting (world/background lighting), the diffuse (directional light impact), specular (the glossiness of an object).
  Test: Check if lighting behaviour was correct on a cube shape
- Abstraction of the “material” of an object which includes all the previous attributes mentioned to make lighting calculations easier to interpret.
  Test: Check if lighting behaviour was correct on a cube shape
- Support for diffuse maps and specular maps. These are essentially images that dictate the diffuse or specular value of an object at a specific coordinate. Again, these will not supported by simplified meshes so will not be used, however implemented for possible future work.
  Test: Check if cube loaded with correct textures
- Logic for directional lights, point lights and spotlights included. Directional lights, such as the sun, are ones where the light rays are parallel to each other and uniform throughout the scene. Point lights, such as light bulbs, involve rays that radiate in a sphere around a point and gradually decrease in its intensity. Spotlights, such as flashlights, whose rays are spread in a cone shape from the source.
  Test: Check if all types of light work as intended on the cubes, check if the addition of multiple lights is accounted for

3.1.3 Model Loading

- A mesh class was constructed which defined each mesh to have vertices, indices (of vertices that make up the faces), textures (for future use). The draw function to handle the drawing of the mesh to the screen, including setting up the relevant attributes and buffers was done.
  Test: Insert cube shape (including positions and indices) into the mesh class and see if the draw function correctly displays the cube
- Implementation of Assimp library to extract .obj file data including vertices, edges and faces into a Model class. The model class is made of 1 or more meshes. The .obj files that I downloaded consist of 1 full mesh.
  Test: Attempt to load bunny.obj into the model class, which will handle the extraction of data and draw to the screen.

3.1.4 OpenMesh

- Creating a MyOpenMesh class that handled calling OpenMesh's QEM simplification functions, with control over the number of vertices. OpenMesh had its own loading of an obj in its own data structures for simplification. It uses the half-edge mesh representation for easy traversal and manipulation of the mesh. After simplification, it writes the simplified mesh's data to another obj file (essentially exporting it) to simplified_mesh.obj.

  Test: Load the bunny.obj file, simplify it to a 1000 vertices and see the result. Activate wireframe to confirm the lower of number of polygons.

3.1.5 ImGui

- Next, the user interface class MyImGui was created and menus were designed. These menus included showing the controls, importing window and mesh information (vertex count, face count etc.) window.

  Test: Run application, check whether ImGui menus popped up correctly, check if the mesh information was correct.
- Implementation of "browse" button which would show an ImGui file browsing menu, in which the user can select a .obj file to load in.

  Test: Check if window shows correctly, and navigation of directory is correct. Confirm that the selected .obj file is loaded in.
- Slider to adjust the desired vertex count for the simplified mesh. "Simplify" button which when pressed, should call the MyOpenMesh function of simplification to the selected vertex count.

  Test: Check that when the button is pressed, the simplification process is underway (printing to the terminal to confirm this), and after the process, the simplified mesh is showing next to the original mesh. Toggling wireframe mode to confirm that the polygon has reduced

3.1.6 My Implementation of the QEM

- Function to open and extract the relevant data from .obj files, namely the positions and normals.

  Test: Compare Mesh object created using my function and the one Assimp produced. They should both show exactly the same
- Creating a HalfEdge struct representing each half edge of the mesh. This included its vertex, a pointer to the next half edge, its twin, the face it belongs to, and a cost parameter. Populating the vector of HalfEdges which contains all half edges of the mesh.

  Test: After extraction into the vector, output the vertices linked to each half edge and compare this to the raw .obj file (opened in a text editor) to confirm if the half edges

are properly linked. Unfortunately, this function did not work as intended, which will addressed later

- Function to calculate the fundamental quadrics of each vertex. See the 2.2. QEM section for details.

  Test: Manually calculate the quadric 1 particular vertex using the raw .obj file, and comparing this against the output of the function
- Function to find the least cost edge. See QEM section for details.

  Test: Manually calculate the cost for 1 particular edge in the mesh using the raw .obj file, and comparing this against the output of the function
- Function to collapse the edge with the least cost. Since there was no code implementation included in the QEM paper (Michael Garland, 1997), I had to think about the connectivity logic myself. For the given edge to collapse with vertices $\mathbf{v_1}$ and $\mathbf{v_2}$ the algorithm works as follows (see Figure 9 for a visual representation):
  1) Make $\mathbf{v_1}$'s position to be the midpoint of the edge and update its quadric to be the sum of $\mathbf{Q_1}$ and $\mathbf{Q_2}$
  2) Mark the faces linked to the half edge and twin half edge to be removed
  3) Iteratively change the connections of edges connected to $\mathbf{v_2}$ in a clockwise fashion until all have been accounted for
  4) Make sure that the relevant half edges' face, next and twin half edge are updated correctly
  5) Finally pairing the new twins together

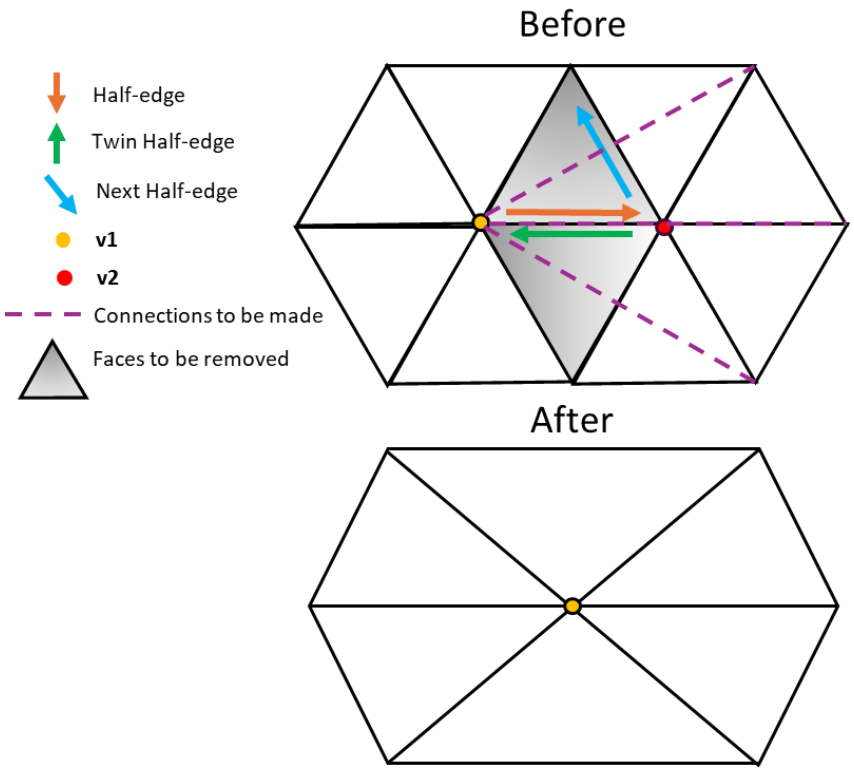


Figure 9: QEM Implementation Diagram

- Function to delete all faces and edges that have been marked for deletion.
  Test: Output number of edges and faces to the console, to check if exactly 2 faces were removed
- Function to simplify the model which aggregates all the previous functions in the correct order and iteratively decimates the mesh 1 vertex at a time until the desired vertex count is reached.
  Test: Compare the simplified model with the original, check the mesh info window for the correct vertex count displayed, check wireframe mode to confirm simplified version is made of less polygons

Parts of my algorithm did function as intended including the extraction of data from the .obj file, the calculation of quadrics and the calculation of the least cost edge. Unfortunately, when creating the half edges, there seemed to be certain ones that did not have a twin edge, which led to errors being generated when traversing through the mesh. Given the time I had left, this issue was not solved, which in turn meant I could not verify whether the collapsing edge function or the deletion of edges and faces function worked correctly. Luckily, the fallback plan of sticking with OpenMesh's implementation was still an option, and one I decided to use.

3.2 User Feedback and Evaluation

From continuous user feedback during the project implementation phase, certain aspects of this application were changed.

The position of the slider to change desired vertex count was adjusted. Users stated this was too small of a slider to control vertex count properly. This was changed to make the slider a lot wider (see Figure 10).

Showing percentage of the simplified model compared to the original model was an added feature. Users stated they wanted an easier way to determine the relative size of the simplified model compared to the original model. This was achieved by showing the percentage of vertices left of the original model (see Figure 10).

Controls were made to show and hide menus. Users stated the menus would get in the way of viewing the models fully. This was remedied by adding controls to show/hide the menus

Immediate simplification on slider value change was implemented. Users wanted instant simplification as soon as the slider value was changed instead of clicking a simplify button every time. This was achieved by simplifying the mesh if the mouse was released after changing the slider value. The button was removed (see Figure 10).

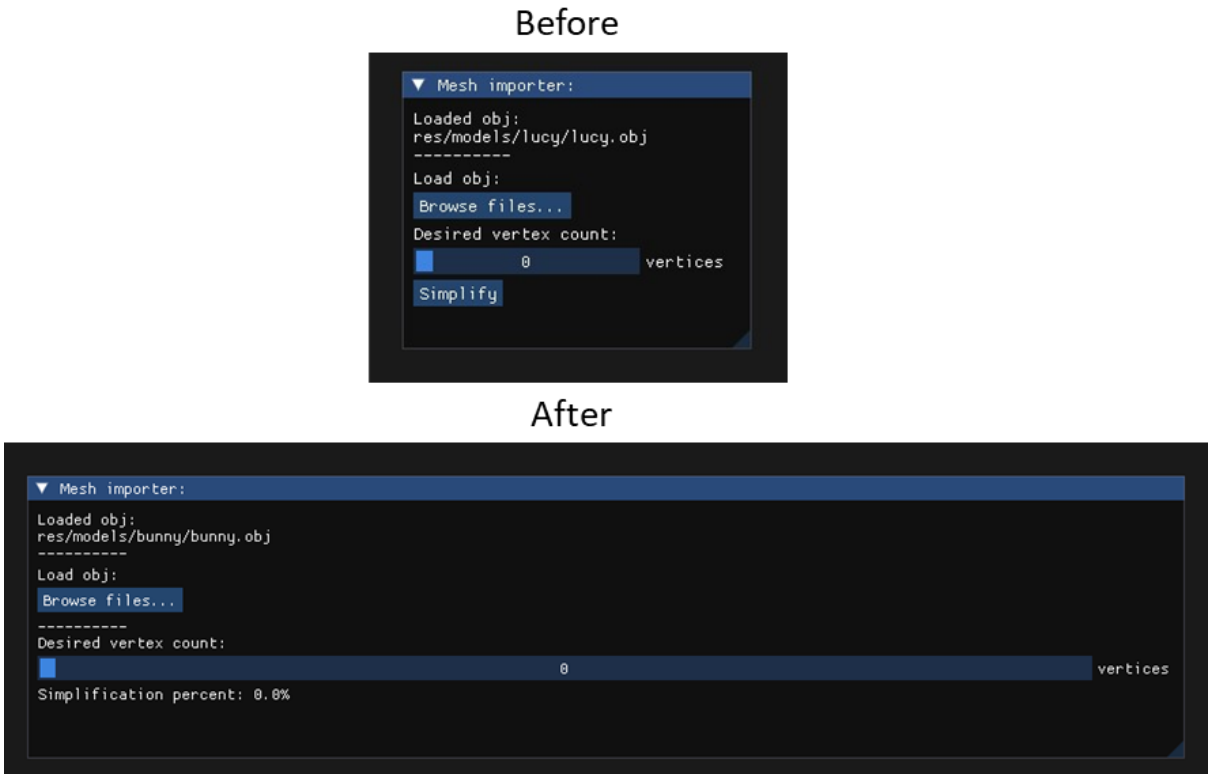


Figure 10: Changes to UI

From the questionnaire (shown under the Appendix section), data was gathered including basic information such as the OS they used, experience on downloading and using the application and whether the application was useful.

Improvements to the GitHub page were recommended including the presentation of the README file and instructions in-case the app doesn't run properly.

Users would have liked some colour, maybe even textures on the models. One user had MacOS which meant the application didn't work for them. It would be better if this was made compatible to MacOS and Linux. Another individual kept getting errors when running the application since I accidentally distributed the debug build rather than the release build.

3.3 Results Analysis

Metrics including the time taken to simplify the mesh to the given vertex count, the time taken to draw the model to the screen, and a self-assigned preservation value were taken. The preservation value was assigned according to my own judgement of the features of the original model that were preserved, 1 representing poor results, 10 representing perfect results. Below are the metrics for two models, the Lucy and Bunny model.

Lucy	Vertex count	Time taken to simplify mesh	Time taken to draw to screen	Draw time difference	Feature preservation

		(milliseconds/ms)	(microseconds/us)	compared to 100%	(scale 1 - 10)
Lucy 100%	33323	N/A	56	N/A	N/A
Lucy 50%	16662	129	43	13	9
Lucy 25%	8330	174	23	33	7
Lucy 10%	3316	201	15	41	5
Lucy 5%	1658	216	13	43	3
Lucy 1%	332	226	13	43	2

Table 1: Results - Lucy Model

Bunny	Vertex count	Time taken to simplify mesh (milliseconds/ms)	Time taken to draw to screen (microseconds/us)	Draw time difference compared to 100%	Feature preservation (scale 1 - 10)
Bunny 100%	23210	N/A	111	N/A	N/A
Bunny 50%	11604	91	68	43	9
Bunny 25%	5802	124	34	77	8
Bunny 10%	2338	146	20	91	7
Bunny 5%	1154	151	15	96	6
Bunny 1%	230	160	14	97	3

Table 2: Results - Bunny Model

We can notice that the time taken to simplify a mesh increases as the number of vertices decrease, which is expected since incremental decimation removes 1 vertex at a time. However, the rate of change in time decreases as the number of vertices decrease also. This is due to the algorithm having to calculate a significantly lower number of costs for edges when vertex count is low. It could also be due to each vertex having a lower valence (number of edges it connects to) when there are less vertices, which means there would be less connections to change.

The time taken to draw to the screen decreases as the number of vertices decrease, again expected since a lower triangle count would require less draw calls to the GPU.

What is notable is the feature preservation values, which indicate that there is not a standard percent decrease that would suit every model. This can be see when the Lucy model at 10% has a preservation of 5, which most likely would not be suitable to use, however, the bunny model at 10% has a value of 7, which would still be appropriate as the final model in the user's desired world. Of course this is a subjective metric and so the user can decide exactly what percent best suits their needs.

3.4 Ideas for Future Work

Including support for colours and textures seems an obvious future goal, not only for the original mesh, but for the simplified mesh, since there exist techniques to simplify textures.

Another feature to add would be allowing various file formats to run within my application including the popular .dae, .fbx and .stl formats.

Integrating further mesh simplification techniques into the application would be a great addition. This would allow users to choose between different simplification methods which may be more suited to their required goals.

Quality of life features such as having 2 separate cameras or fixed cameras for visualisation purposes, and rotation of models could be added. This would make it easier for the user to navigate and inspect the models.

Adding extras such as preset background scenes could aid the user in visualising how their models would look after exporting these models into their world.

List of References

[1] Garland, M. and Heckbert, Surface Simplification Using Quadric Error Metrics. [online] Available at: http://www.mgarland.org/files/papers/quadrics.pdf

[2] Code Envato Tuts+. (2012). Collision Detection Using the Separating Axis Theorem | Envato Tuts+. [online] Available at: https://code.tutsplus.com/collision-detection-using-the-separating-axis-theorem--gamedev-169t

[3] Baumgart, B. (1972). Winged Edge Polyhedron representation. [online] Available at: http://i.stanford.edu/pub/cstr/reports/cs/tr/72/320/CS-TR-72-320.pdf

[4] Tobler, R. and Maierhofer. (2006). A Mesh Data Structure for Rendering and Subdivision. [online] Available at: http://wscg.zcu.cz/wscg2006/Papers_2006/Short/E17-full.pdf

[5] Helton H. Biscaro, Farima L.S. Nunes. (2016). Comparing efficient data structures to represent geometric models for three-dimensional virtual medical training. [online] Available at: https://www.sciencedirect.com/science/article/pii/S153204641630096X/pdfft?md5=47c9dd3f63c7851d7810dc5427e63746&pid=1-s2.0-S153204641630096X-main.pdf

[6] Spatial Team. What is hybrid modelling? [online] Available at: https://www.spatial.com/resources/glossary/what-is-hybrid-modeling

[7] Mario Botsch, Leif K.. (2010). Polygon Mesh Processing: Simplification and Approximation. [online] Available at: https://www.pmp-book.org/

[8] M. Eck, T. DeRose, T. Duchamp. (1995). Multiresolution Analysis of Arbitrary Meshes. [online] Available at: https://dl.acm.org/doi/10.1145/218380.218440

[9] William J. Schroeder, William E. Lorensen, Jonathan A. Zarge. (1992). Decimation of Triangle Meshes. [online] Available at: https://dl.acm.org/doi/10.1145/142920.134010

[10] Joey De Vries. (2020). Learn OpenGL: Graphics Programming. [online] Available at: https://learnopengl.com/

Appendix A
Self-appraisal

A.1 Critical self-evaluation

This project provided an opportunity to enhance my technical skills and become more proficient in C++ and OpenGL, as well as giving a broader understanding of how graphics APIs work.

It allowed me to improve the use of tools that are widely used by professionals, including GitHub for version control and Visual Studio for large development projects.

Time management was a key area for improvement. There were plenty of instances where I underestimated the complexity of the task at hand, leading to delays which eventually led to certain aims not being met.

More rigorous testing procedures could have been implemented, which would not only save crucial time, but also ensuring the application doesn't break under extraneous situations.

Setting realistic goals and deadlines is something needed to be worked on.

The project process provided valuable learning experiences for my personal and professional growth. It has highlighted key areas in which improvement is necessary for a prosperous career in the computer science field.

A.2 Legal, social, ethical and professional issues

A.3.1 Legal issues

Ensuring compliance with licensing agreements and copyright laws was taken into account during the project. Necessary when using third party libraries and external material, such as images, figures and models.

A.3.2 Social issues

Social issues were not relevant to this project.

A.3.3 Ethical issues

An ethical issue handled well was respecting the intellectual property rights when using external material.

A.3.4 Professional issues

When communicating with stakeholders, I made sure it was in a respectful manner.

Appendix B
External Materials

Figure 2 image - https://download.autodesk.com/global/docs/softimage2014/en_us/index.html?url=files/poly_basic_PolygonMeshes.htm,topicNumber=d30e91379

MeshLab - https://www.meshlab.net/

Simplygon - https://www.simplygon.com/

Blender - https://www.blender.org/

OpenGL- https://www.opengl.org/

Unreal Engine - https://www.unrealengine.com/en-US

Unity - https://unity.com/

Stackscale - https://www.stackscale.com/blog/most-popular-programming-languages/

Statista - https://www.statista.com/statistics/268237/global-market-share-held-by-operating-systems-since-2009/

Steam charts - https://store.steampowered.com/hwsurvey/steam-hardware-software-survey-welcome-to-steam

Khronos group - https://www.khronos.org/

Visual Studio - https://visualstudio.microsoft.com/

Microsoft - https://www.microsoft.com/en-gb/

Vulkan guide - https://vkguide.dev/docs/new_chapter_1/vulkan_command_flow/

DirectX - https://www.microsoft.com/en-gb/download/details.aspx?id=35

Vulkan - https://www.vulkan.org/

OpenGL - https://www.opengl.org/

ImGui - https://www.dearimgui.com/

Qt - https://www.qt.io/

Stanford 3D scanning repository - https://graphics.stanford.edu/data/3Dscanrep/

GitHub repository with .obj files - https://github.com/alecjacobson/common-3d-test-models?tab=readme-ov-file

GLFW - https://www.glfw.org/

GLAD - https://glad.dav1d.de/

Dirent.h - https://github.com/tronkko/dirent/tree/master/include

ImGui file dialog - https://github.com/aiekick/ImGuiFileDialog?tab=MIT-1-ov-file

Appendix C
Supporting Materials

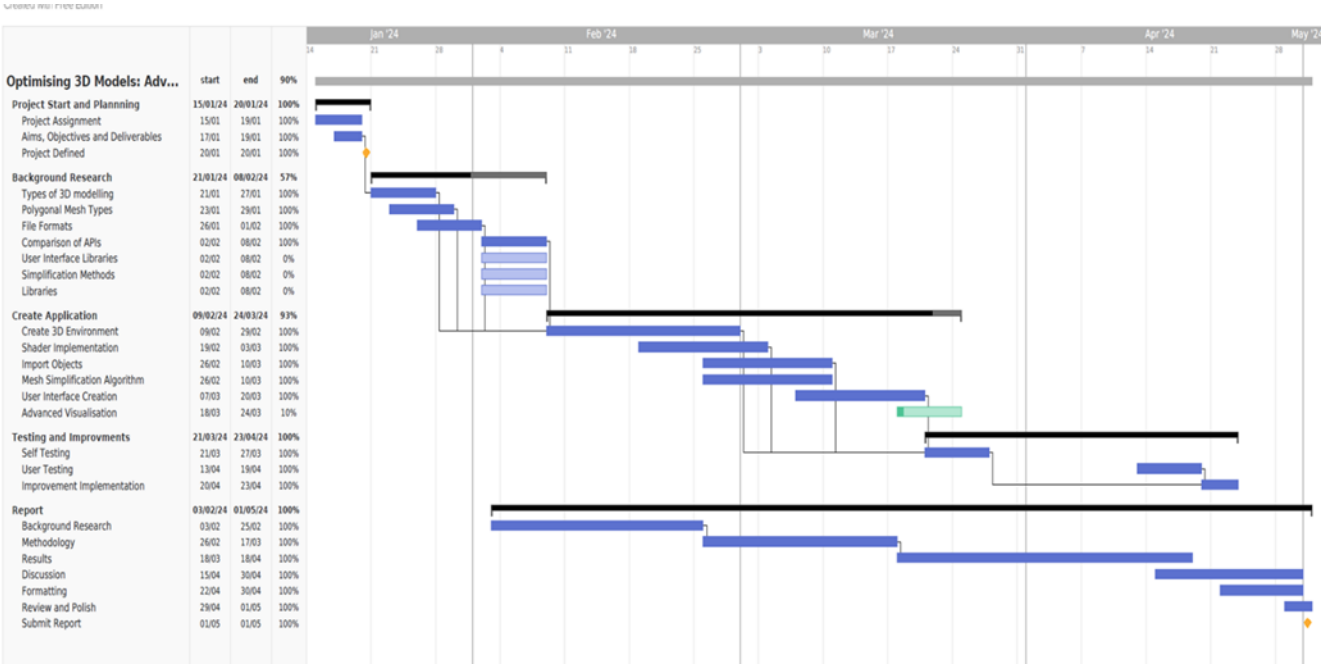


Figure 11: Gantt Chart

Mesh Simplification Application Questionnaire

This questionnaire is to support a project to design and create an application that allows mesh simplification. The aim is to allow visualisation of mesh simplification for the user and comparison to the original object. The other aim of the this application is to allow users to import an object of their choice and export the reduced object for their own use. The questionnaire is to support the testing of the application and receive any feedback for improvement.

This questionnaire is anonymous and can be filled out by anyone. The data collected shall be handled with care and shall not infringe the privacy of an individual. The answers you will not be linked with your account or identity. Due to the anonymity of the responses, once submitted, it is not possible to withdraw consent for responses being used.

* Indicates required question

1. What is your age range? *

Mark only one oval.

- ◯ 0 - 18
- ◯ 19 - 25
- ◯ 26 - 30
- ◯ 31 - 40
- ◯ 41 - 60
- ◯ 61+

2. Do you have Windows Operation System? *

Mark only one oval.

- ◯ Yes
- ◯ No (If No then please skip all remaining questions and submit response. This will help understand the % of people with Widnows and % with other operating systems

Links to access GitHub page for application and how-to guide

Link to the GitHub page: https://github.com/SavanHathalia/3D-Mesh-Simplification

Please click on the link and open up the "READ ME" file. This will provide guidance on downloading and using the application.

3. Did you manage to access the GitHub page and download the application successfully?

Mark only one oval.

- ◯ Yes
- ◯ No

4. Does the application work when opened?

Mark only one oval.

- ◯ Yes
- ◯ No

5. Are you able to import an object successfully?

Mark only one oval.

- ◯ Yes
- ◯ No

6. Are you able to run the mesh simplification on the imported model?

Mark only one oval.

- ◯ Yes
- ◯ No

7. Are the mesh sizes suitable?

Mark only one oval.

- ◯ Yes
- ◯ No

8. Are you able to export your model?

Mark only one oval.

- ◯ Yes
- ◯ No

9. How useful are the instructions on the GitHub page?

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

10. Are there any improvements that can be made to the instructions on the GitHub page?

11. How useful has the application been?

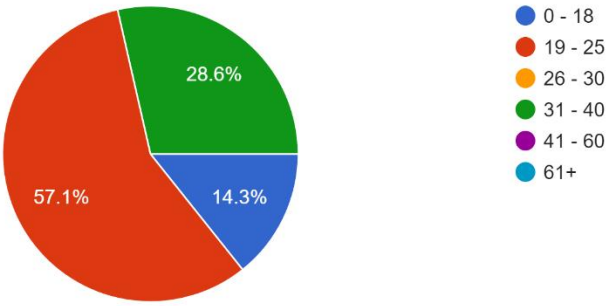
Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very useful

12. Are there any improvements that can be made to the application?

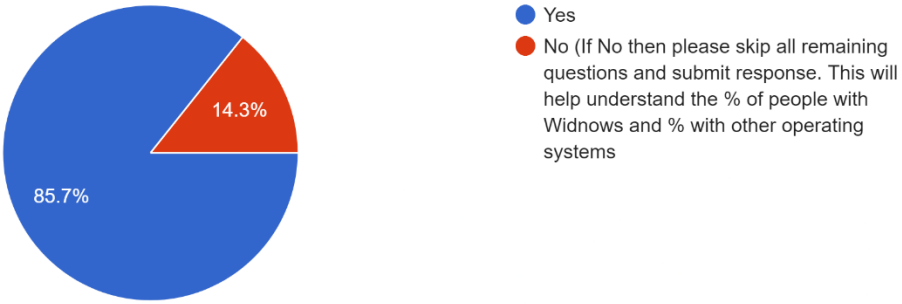
What is your age range?

7 responses



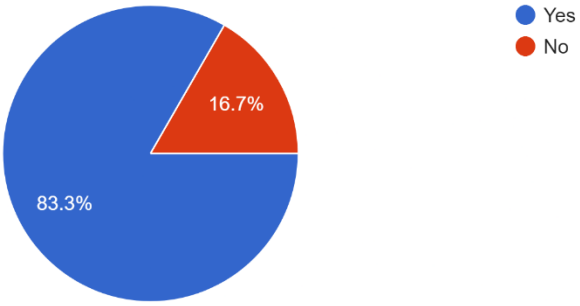
Do you have Windows Operation System?

7 responses



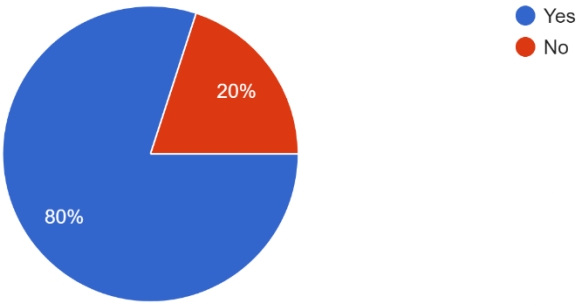
Did you manage to access the GitHub page and download the application successfully?

6 responses



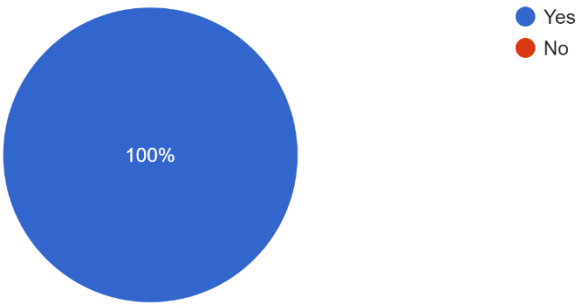
Does the application work when opened?

5 responses



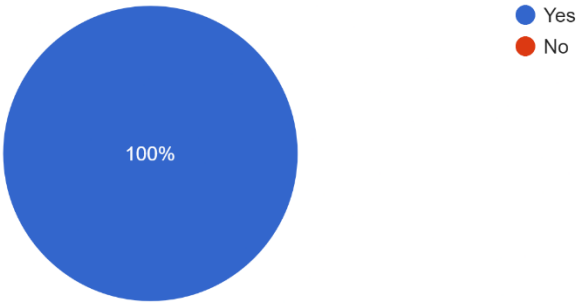
Are you able to import an object successfully?

4 responses



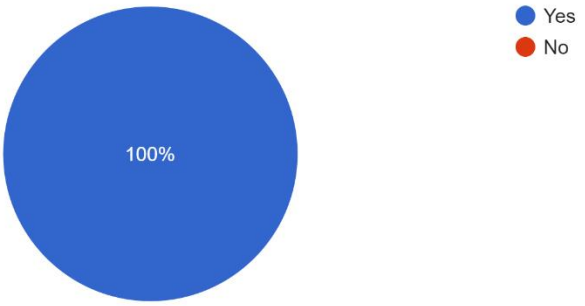
Are you able to run the mesh simplification on the imported model?

4 responses



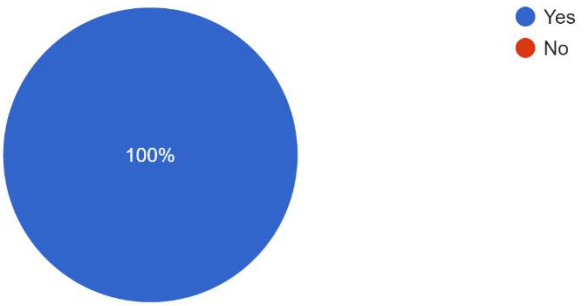
Are the mesh sizes suitable?

4 responses



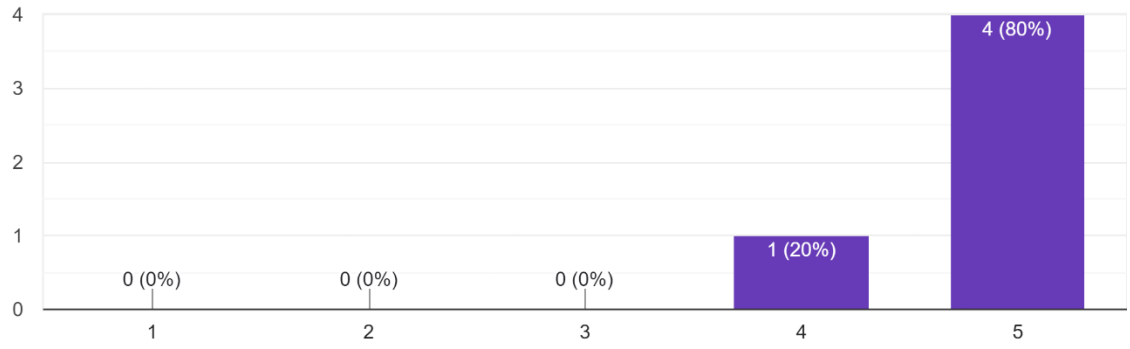
Are you able to export your model?

4 responses



How useful are the instructions on the GitHub page?

5 responses



Are there any improvements that can be made to the instructions on the GitHub page?

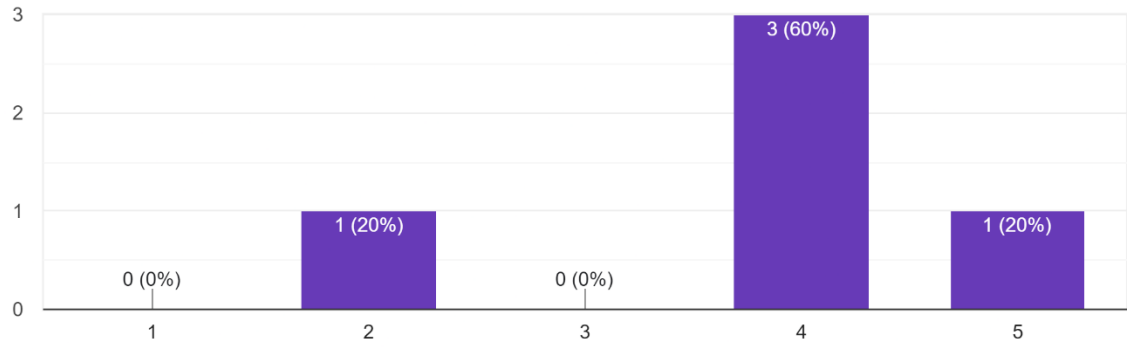
2 responses

Excellent instructions and walked me straight through the process. Improvement could be the presentation of the READ ME file.

instruction for if something goes wrong could be helpful as the app didn't work as intended for me.

How useful has the application been?

5 responses



Are there any improvements that can be made to the application?

5 responses

Application worked well. The good thing was that I was able to the imported model with the meshes and then a copy was made to show the reduce model mesh. That really showed the differences. I would only recommend adding some colours to the meshes.

make the application for mac

Mesh could be better presented. Maybe give option to change the mesh shapes.

was not able to download the application. Kept getting an error

Add textures so that the models created are better

Link to GitHub Repository: https://github.com/SavanHathalia/3D-Mesh-Simplification/tree/main

Link to video demonstration: https://www.youtube.com/watch?v=8PXvJByTtBw