Tampere University

Antti Heinonen

SIMPLIFICATION OF 3D COMPUTER-AIDED DESIGN MODELS TO IMPROVE RENDERING PERFORMANCE

Faculty of Information Technology and Communication Sciences
Master of Science Thesis
January 2019

ABSTRACT

ANTTI HEINONEN: Simplification of 3D Computer-aided Design Models to Improve Rendering Performance
Tampere University
Master of Science thesis, 57 pages, 9 Appendix pages
January 2019
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiners: Assistant Prof. Pekka Jääskeläinen, M.Sc. Matias Koskela
Keywords: CAD, 3D model, virtual reality, mobile, augmented reality, simplification, rendering, performance

Visualization of three-dimensional (3D) computer-aided design model is an integral part of the design process. Large assemblies such as plant or building designs contain a substantial amount of geometric data. New constraints for visualization performance and the amount of geometric data are set by the advent of mobile devices and virtual reality headsets. Our goal is to improve visualization performance and reduce memory consumption by simplifying 3D models while retaining the output simplification quality stable regardless of the geometric complexity of the input mesh.

We research the current state of 3D mesh simplification methods that use geometry decimation. We design and implement our own data structure for geometry decimation. Based on the existing research, we select and use an edge decimation method for model simplification. In order to free the user from configuring edge decimation level per model by hand, and to retain a stable quality of the simplification output, we propose a threshold parameter, *edge decimation cost threshold*. The threshold is calculated by multiplying the length of the model's bounding box diagonal with a user-defined scale parameter.

Our results show that the edge decimation cost threshold works as expected. The geometry decimation algorithm manages to simplify models with round surfaces with an excellent simplification rate. Based on the edge decimation cost threshold, the algorithm terminates the geometry decimation for models that have a large number of planar surfaces. Without the threshold, the simplification leads to large geometric errors quickly. The visualization performance improvement from the simplification scales almost at the same rate as the simplification rate.

TIIVISTELMÄ

ANTTI HEINONEN: Kolmiulotteisten tietokoneavusteisten mallien yksinkertaistaminen renderoinnin nopeuttamiseksi
Tampereen yliopisto
Diplomityö, 57 sivua, 9 liitesivua
Tammikuu 2019
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Apul.prof. Pekka Jääskeläinen, DI. Matias Koskela
Avainsanat: CAD, 3D malli, yksinkertaistaminen, mobiili, virtuaalitodellisuus, lisätty todellisuus, renderointi, tehokkuus

Kolmiulotteisten (3D) tietokoneavusteisten mallien visualisointi on tärkeä osa suunnitteluprosessia. Isot kokoonpanot, kuten laitos- ja talomallit, sisältävät suuria määriä geometrista tietoa. Mobiililaitteiden ja virtuaalitodellisuuden lisääntyvä käyttö visualisoinnin apuna asettaa uusia rajoitteita visualisoinnin tehokkuudelle, sekä geometrisen tiedon määrälle. Tavoitteenamme on parantaa visualisoinnin tehokkuutta ja pienentää muistinkulutusta yksinkertaistamalla 3D malleja, pitäen samalla ulostulevan yksinkertaistuksen laadun tasaisena riippumatta syötteenä annetun mallin geometrisesta monimutkaisuudesta.

Tutkimme 3D mallien yksinkertaistamisen tapoja, jotka käyttävät geometrian poistoa. Suunnittelemme ja toteutamme oman tietorakenteen geometrian poistoa varten. Perustuen olemassa olevaan tutkimukseen, valitsemme ja käytämme särmien poistoa mallin yksinkertaistamiseksi. Vapauttaakseen käyttäjän määrittelemästä särmien poiston määrää erikseen jokaiselle mallille, ehdotamme kynnysarvoparametria, *särmän poiston kustannuksen kynnysarvo*. Kynnysarvo lasketaan kertomalla mallin rajaavan laatikon lävistäjän pituus ja käyttäjän määrittämä skaala-arvo.

Tuloksemme näyttävät, että särmän poiston kustannuksen kynnysarvo toimii odotetusti. Geometrianpoistoalgoritmi onnistuu erinomaisesti yksinkertaistamaan malleja, jotka sisältää paljon pyöreitä pintoja. Algoritmi pysäyttää geometrian poiston paljon suoria pintoja sisältäville malleille. Ilman kynnysarvoa, yksinkertaistaminen johtaa nopeasti suureen geometriseen virhetasoon. Yksinkertaistamisesta johtuva visualisoinnin tehokkuuden parannus skaalautuu melkein samassa suhteessa kuin yksinkertaistamisen taso.

PREFACE

I would like to thank Vertex Systems Oy for providing the idea and giving me time and space to complete this thesis. Although the work on this thesis was carried by me alone, I would like to thank my colleagues at Vertex Systems for ideas, thoughts, and guidance during the process. I would like to thank the examiners of this thesis, Assistant Prof. Pekka Jääskeläinen and M.Sc. Matias Koskela for providing valuable feedback and guidance.

Finally, I would like to thank those who are close to me, who supported me during the thesis project and my studies at the Tampere University of Technology.

Tampere, 11.1.2019

Antti Heinonen

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

3D	Three-dimensional
AR	Augmented reality
B-rep	Boundary representation
CAD	Computer-aided design
CSG	Constructive solid geometry
fps	Frames per second
glTF	GL Transmission Format
GPU	Graphics processing unit
LOD	Level-of-detail
MR	Mixed reality
OCAT	The Open Capture and Analytics Tool
QEM	Quadric error metric
RMS	Root mean square
smf	Simple model format
VR	Virtual reality

1. INTRODUCTION

In this thesis, we focus on reviewing techniques for simplifying large virtual industrial *three-dimensional* (3D) design models, such as building or plant designs. We develop an offline simplification method based on existing research. Using the method, we measure rendering time performance improvements in desktop and mobile environments. Currently, the amount of geometry data in models that are used to visualize the design on a desktop computer is too significant to be presented efficiently in a virtual reality environment on a desktop computer or mobile technology.

This work is done for Vertex Systems Oy which provides *computer-aided design* (CAD) and product data management software for various fields of industry. Major industries include building, machinery, plant, and interior industry. The numerous different types of designs imply that the resulting implementation cannot be focused on the design of a single type. The implementation is planned to be integrated into Vertex CAD products in the future. Figure 1.1 illustrates various industrial models designed using Vertex CAD products.

In order to use a computer to design arbitrary items such as machinery, buildings, and parts of which such structures are assembled, CAD software is used. A 3D representation is used to visualize an item to aid the design work. The 3D representation which is a result of the design is called a *3D model*. The process of designing items using 3D CAD software is called *modeling*. A 3D representation is an integral part of CAD software as it helps the designer to view the design model virtually.

Precision in CAD software is vital as the goal is to design physically plausible items that can be manufactured in the real world. In order to achieve such accuracy, mathematical models are used to represent the 3D model. Mathematical models guarantee continuous accuracy of 3D model surfaces when they are modified or measured. As such mathematical models are complicated to solve, an estimation of the mathematical model is created to visualize the model in real time. A *polygon mesh* is created from the mathematical model with precise accuracy to accelerate the visualization and to take advantage of the graphics hardware. The accuracy of the mesh is a compromise between visualization performance and visual quality.

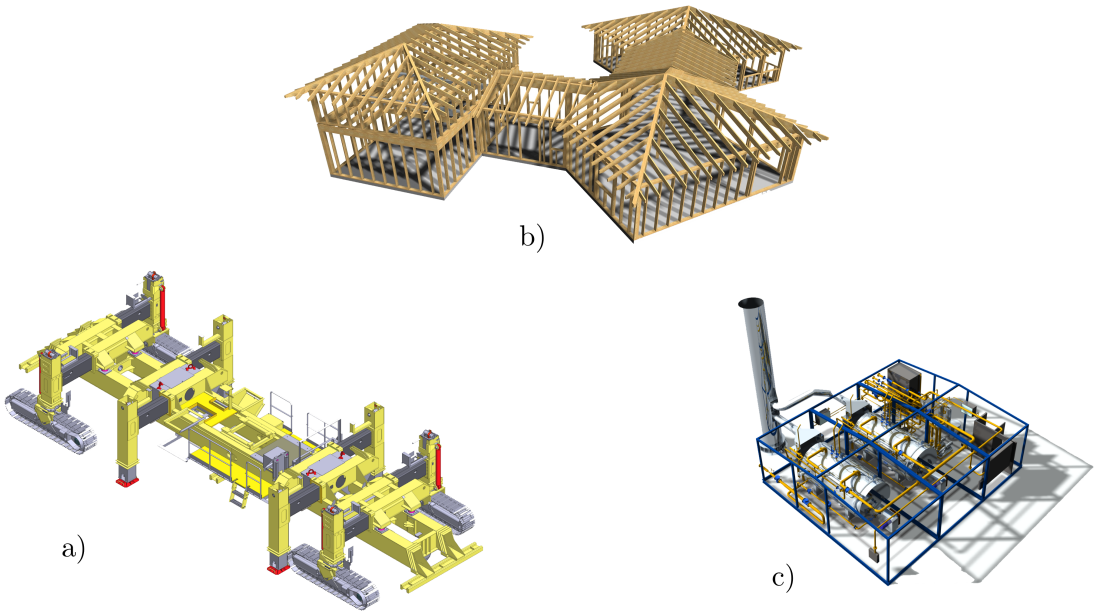


Figure 1.1 *Models designed using Vertex CAD software: a) mechanical assembly b) building framing c) plant design. [1]*

The polygon mesh is a discrete representation of the underlying mathematical model consisting of points, edges, and faces.

In recent years, the advances in computational performance have enabled new ways to visualize 3D models. In this thesis, we focus on measuring the performance improvements of simplified 3D models in desktop and mobile environments. On both environments, *virtual reality* (VR) technology is exciting as it allows the 3D model to be experienced in the real world scale, that further helps a viewer to perceive how the design displays in the real environment. In addition to VR, mobile technology allows, for example, using *augmented reality* (AR) to visualize the 3D model mixed to a real environment. Milgram et al. define the mixing of the real-world and virtual environment as a virtuality continuum [2]. A continuum that covers real environments and completely virtual environments. The paper defines term *mixed reality* (MR) which refers to display techniques placed within the virtuality continuum.

Even though computational performance has allowed such visualization techniques, still limitations exist how complex 3D meshes can be handled in VR or mobile in comparison to a desktop computer with a dedicated graphics card. In order to visualize a complex mesh in VR and mobile environments, we are faced with an optimization problem; a need to reduce the complexity of a 3D mesh to a level where it can be effectively visualized in mobile and VR environments. The optimization

problems raise new challenges, such as how much a 3D mesh can be simplified while maintaining an acceptable visual level and the final visualization performance.

In a 3D CAD model simplification report made in 2015 [3], Arvo et al. research, describe and test common simplification algorithms for CAD mesh model data. In their survey, they are faced with a similar problem as we do; how mesh simplification algorithms are used to simplify highly detailed CAD models to improve performance and reduce memory requirements to be used in augmented reality applications on mobile.

The remainder of this thesis is organized as follows. In Chapter 2 we introduce basic theory of 3D models and build the background for mesh simplification. Chapter 3 continues with it, and we explore previous work and different methods of mesh simplification by geometry decimation. In Chapter 4, we explain the details of our mesh simplification implementation and the simplification method that is used. In order to handle meshes with different complexities, we propose an *edge decimation cost threshold* to guide the decimation. In Chapter 5, we present the results of comparing the output of our simplification method to a similar method, testing the edge decimation cost target against various CAD models, and finally measuring the performance improvements. Finally, we discuss the results and future work in Chapter 6. Appendix A includes results from Hausdorff distance measurement against models decimated using QSlim. In Appendix B images of original and simplified Vertex CAD models are presented.

2. 3D MODEL BASICS

This chapter provides a basic theory for a 3D model, which is an integral part of this work. 3D models are used to represent a three-dimensional object in a virtual environment. The model is either represented analytically by mathematical models or by geometric data, using points, lines, and faces. Figure 2.1 illustrates a visualization of a 3D CAD model build using mathematical models.

Figure 2.1 *Rendering of a 3D CAD model created in Vertex CAD software. [1]*

2.1 Mathematical representation

A 3D model can be based on mathematical surface models, such as *boundary representation* (B-rep), which is a mathematical method to describe curves and surfaces. Another popular mathematical model based technique is *constructive solid geometry* (CSG) where the model is built by combining primitive objects such as spheres and cubes using boolean operations. Figure 2.2 illustrates the difference of these model construction techniques. *Solid modeling* is a method of modeling 3D models using mathematical representations. The point of solid modeling is to ensure the rigidity and 'watertightness' of the model. Stroud explains solid modeling techniques in more detail in his book [4]. Mathematical 3D models are used in software where

the theoretical accuracy of a model is important. For example, in CAD software the designed model is meant to represent the real world equivalent as close as possible. For CAD software, another advantage of solid modeling is the simple and accurate computation of physical properties, such as mass, volume, and center of gravity. To visualize a mathematical model, a *polygon mesh* is usually created from the model. The polygon mesh is an approximation of the underlying mathematical model. If the mesh simplification is performed in the CAD software, we can take advantage of the conversion process. The quality of the mesh approximation could be tweaked to reduce the amount of mesh geometry data to get to the simplification target. In this work, we especially focus on the simplification of polygon meshes created from the mathematical CAD models.

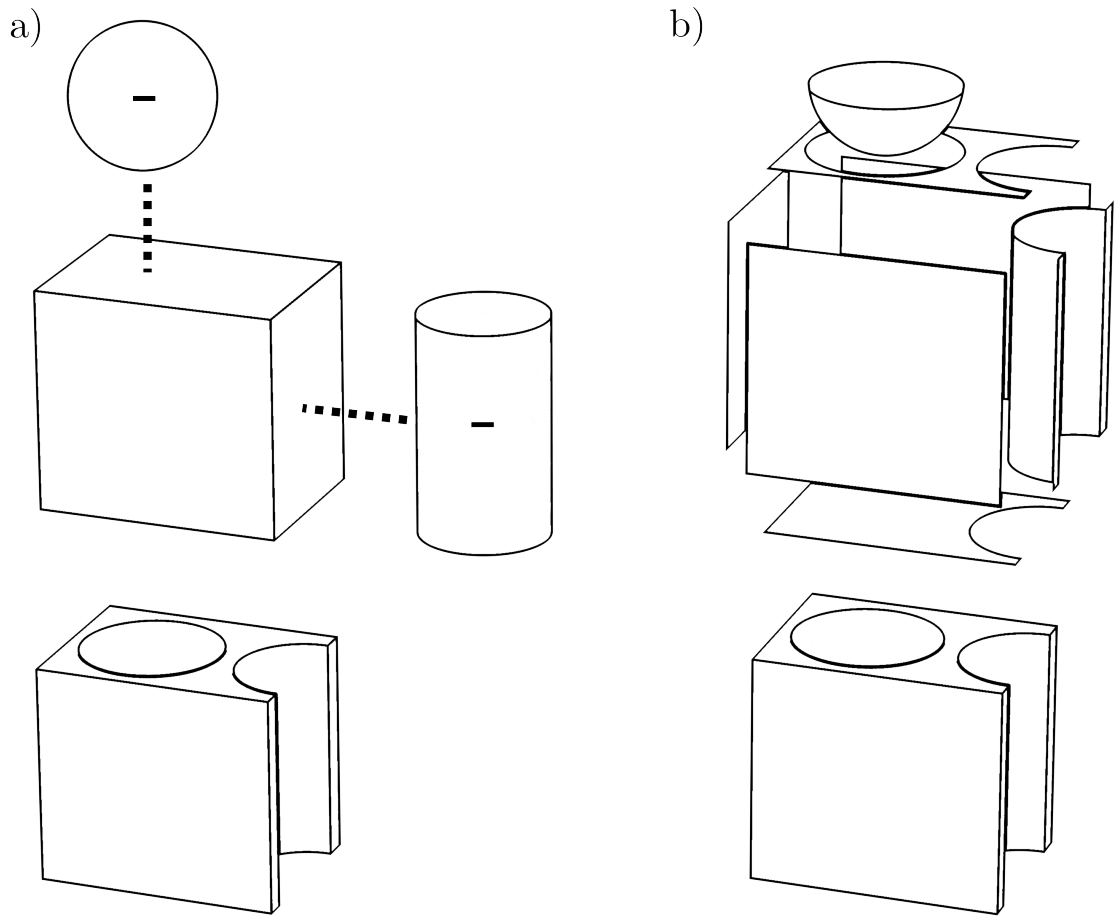


Figure 2.2 *Two methods of solid modeling: a) CSG and b) B-rep.*

2.2 Polygon mesh

As an alternative to a mathematical model, a polygon mesh can be used to represent a 3D model. A polygon mesh consists of vertices and faces. Vertices are three-dimensional points that are connected with faces. A vertex can also contain other attributes such as surface normal vector or texture coordinate to map an image to the surface. A face contains the vertices which define the given face in three-dimensional space. A polygon mesh can be structured in many ways. For example, a *triangle mesh* is popular geometry representation for 3D geometry in hardware accelerated graphics software. In a triangle mesh, a face with triangle type is represented using three vertices. Additionally, vertices can be shared between triangular faces by using a list which contains sets of three indices. Figure 2.3 illustrates how faces, in this case, triangles, form surfaces which in turn create mesh models.

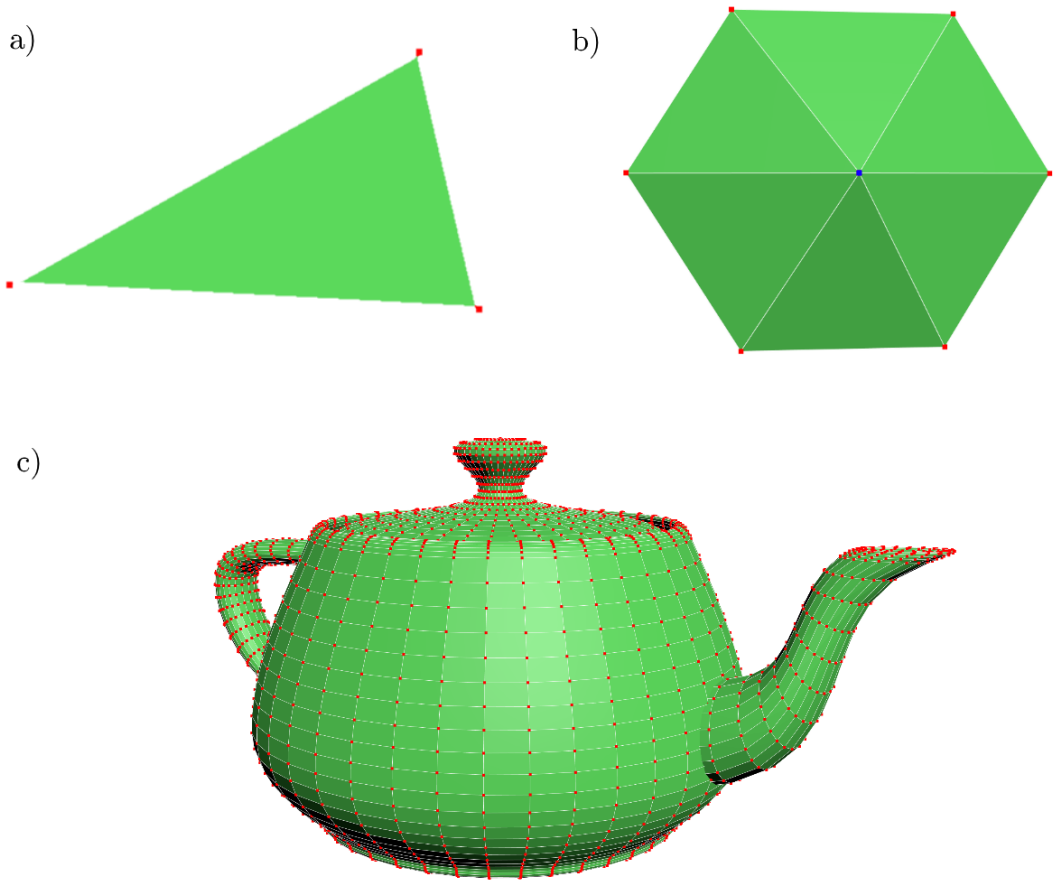


Figure 2.3 *Visualization of a) triangle, b) surface and c) mesh model.*

In comparison to mathematical models, a triangle mesh structure is trivial to render, by iterating the face list and rendering each triangle face as they are read. Polygon mesh structure in which only a list of vertices and indices are available does

not have connections between nearby vertices, edges and triangles readily available. That means that a polygon mesh structure is not optimal data structure when the mesh needs to be modified — for example, using a mesh simplification algorithm. A *half-edge* mesh data structure is one way to address this issue as it has more connectivity information between edges, faces, and points attached to it [5]. The advantage compared to the mathematical model is that the discrete face information is efficiently rasterized by the *graphics processing unit* (GPU) without solving the mathematical model. As triangle mesh is the most popular polygon mesh representation, in this work, we assume that polygon mesh consists only of triangular faces unless otherwise is stated.

We define a polygon mesh $\boldsymbol{M}$ as a pair of vertices $\boldsymbol{V}$ and faces $\boldsymbol{F}$. A vertex list $\boldsymbol{V} = (\boldsymbol{v_1}, \boldsymbol{v_2}, ..., \boldsymbol{v_n})$ is a list of set of attributes $\boldsymbol{v_i} = \{x_p, y_p, z_p, x_n, y_n, z_n, u, v\}$. Where the elements with subscript p represents the vertex position in three-dimensional Euclidean space $\boldsymbol{R^3}$. Elements with subscript n represent a vertex surface normal vector in three-dimensional range $]-1, 1[$ with length of 1. Elements u, v represents a texture coordinate vector at the vertex position and is defined in two-dimensional range $]0, 1[$. A face list $\boldsymbol{F} = (f_1, f_2, ..., f_r)$ is a list of set of indices $f_i = \{j_0, j_1, ..., j_k\}$. In a polygon mesh which has triangular faces, a single face is a set of three indices $f_i = (j_0, j_1, j_2)$. The indices are referring to the vertex list, thus a triangular face $f_i = (j, k, l)$ is defined using vertices $(\boldsymbol{v_j}, \boldsymbol{v_k}, \boldsymbol{v_l})$. Edges of the polygon mesh are implied from face boundaries, for example, a triangular face $f_i = (j, k, l)$ has three edges $e_0 = (\boldsymbol{v_j}, \boldsymbol{v_k}), e_1 = (\boldsymbol{v_k}, \boldsymbol{v_l}), e_2 = (\boldsymbol{v_l}, \boldsymbol{v_j})$.

Mesh topology defines connectivity between the points, forming lines and how lines are connected to form triangles or other polygon faces. A mesh has either a non-manifold or manifold topology space. A mesh has a closed manifold topology space if each edge is related to two faces and every point is part of a closed disk. For example, sphere, cube, and torus have a closed manifold structure. If some edges of the mesh are related to one face only and some points are part of an open disk, then the mesh has a manifold with a boundary. As an example, a plane surface, or a Möbius strip has manifold with a boundary. Figure 2.4 further illustrates the case of non-manifold or manifold topology.

Mesh data structure limits the mesh topology space. For example, a half-edge data structure only supports closed manifold space. Simplification algorithm might modify the mesh topology if the data structure allows, resulting in non-manifold surfaces. That is solved by constraining the simplification to disallow the modification if it results in a non-manifold mesh. If the local surface is non-manifold, the simplification of the region can be omitted. Garland discusses and visualizes the differences

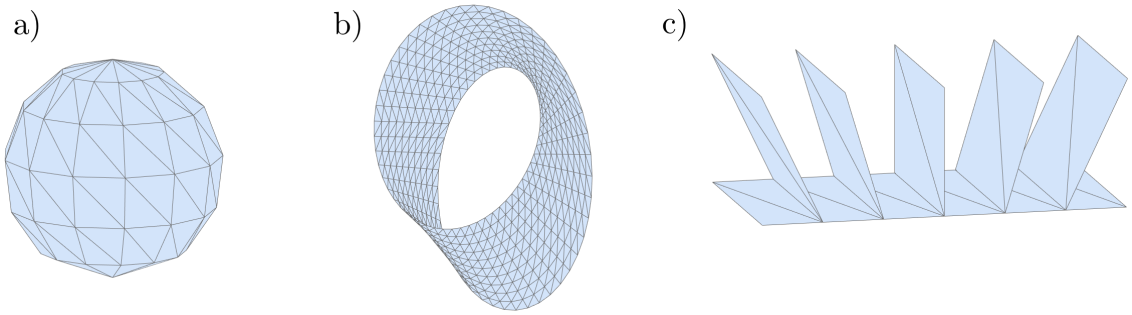


Figure 2.4 *Visualization of a) manifold mesh, a closed sphere, b) Möbius strip, manifold mesh with boundary, c) non-manifold mesh.*

between non-manifold and manifold meshes in great detail in his Ph.D. dissertation [6].

2.3 Rendering

Rendering is a way of visualizing a 3D model. In rendering, a picture is generated from the 3D geometric data using a viewpoint. In short, a renderer transforms the 3D model to a two-dimensional image using a projection. In real-time rendering sequence of pictures, called *frames*, are rendered. The *frame rate* is to measure how many times a frame is rendered per second. Usually, most computer screens update the screen 60 *frames per second* (fps). The time to render one frame is called *frame time*. In a virtual reality environment, a feel of presence is wanted. In order to ensure a good presence, a frame rate of 95 fps should be achieved [7]. On mobile devices, we target a frame rate of 30 fps as it on the vertical blanking interval which is forced on most mobile devices [8, 9]. 30 fps provides acceptable visual perception. In real-time rendered visualization, low latency is wanted if interaction is possible, for example, using a controller. Latency in real-time rendering context means the time from controller input to change on the screen. Low latency is especially important in virtual reality applications where the feeling of presence is dependent on it, as the viewpoint position is updated based on the head position [7].

On current hardware, rasterization is used to perform real-time rendered visualizations of 3D models. Rasterization is a process of transforming a mesh, built of geometry primitives, points, lines, and triangles, directly to a two-dimensional image using transformation matrices and a rasterization algorithm. A rasterization algorithm decides if a given primitive, such as a triangle, fills at currently processed image pixel or row. A rasterization algorithm is either software or hardware based and the algorithm is implementation specific. Current GPUs are designed to pro-

cess, rasterize and shade large amounts of polygon mesh data. Practically every computer has a GPU. Thus it is wise to leverage the additional processing power when rendering a model.

Due to the programmability and parallel nature of GPU, more and more GPUs are used for general-purpose computation tasks. One such task is ray tracing, which has recently gained much momentum as graphics APIs are adding support for it [10, 11]. Furthermore, NVIDIA's Turing architecture added hardware-based ray tracing acceleration, which could mean that in future ray tracing could be a viable method for real-time rendering [12]. Similarly to rasterization, ray tracing can be used to render a mesh. In ray tracing, a mesh is rendered by calculating the nearest intersections from viewport pixels towards a virtual camera's forward direction to the scene space. If ray intersects with a mesh in the scene space, the nearest hit point along the ray relative to the viewpoint is shaded. The shading is often done by ray tracing from the hit point, also known as 'bouncing', to a light source or to some other direction based on a surface shading function. Such function is user-defined and in photo-realistic rendering often based on the physical properties of the surface material. The advantage of ray tracing over rasterization is that it supports programmable ray intersection routines, which means that any arbitrary mathematical model is supported. Additionally, ray tracing is more suited for purposes where high quality or photo-realistic images are required, as ray tracing allows to use more advanced shading techniques. For example, to simulate light transport by mimicking the properties of a photon bouncing around a scene. Currently, as a rendering technique, ray tracing is slower by a large margin compared to rasterization. Deng et al. excellently survey the current state of real-time ray tracing performance [13].

In virtual reality, two viewpoints for each eye have to be rendered in order to create an illusion of presence. Thus during a single frame, the scene has to be rendered twice. Optimization techniques to speed up the rendering has been proposed. For example, instead of rendering each eye in a separate pass, the Unity engine has a single pass rendering mode, where each object is rendered twice during an iteration to a single image which is shared with each eye [14]. That decreases the frame time by minimizing graphics state changes and object visibility checks. As the eye is focused to a single point, foveated rendering has been proposed to address the issue by focusing the pixel processing in the center of the viewpoint. That is, rendering the inset of the viewpoint using higher resolution and the outset by lower resolution [15].

2.4 Mesh complexity

Polygon meshes can have different levels of detail. Mesh *level-of-detail* (LOD) is used to describe the visual quality of the mesh, that is, how closely the mesh is representing the model. In Figure 2.5 three different level-of-detail meshes from the same base mesh are displayed. Usually when high fidelity is wanted a high polygon count meshes are used. The perception of visual quality of a mesh is affected by the distance between the viewpoint and the mesh. For example, if the mesh is rendered distant from the viewpoint where the mesh appears small, many polygons are rasterized and shaded per viewpoint pixel, which leads to unnecessary computation. That problem is often alleviated by having multiple level-of-detail versions of the mesh and changing it based on the distance to the viewpoint. The level of detail, or in other words, polygon density often changes locally to the mesh itself. Different surface areas of the mesh usually differ in polygon density. Local polygon density becomes important in mesh simplification, as it is beneficial to remove polygons from areas where high polygon density is unnecessary. For example, in flat surface areas where removing polygons do not affect the quality of the mesh as much as areas that have irregular or round surfaces, such as curves.

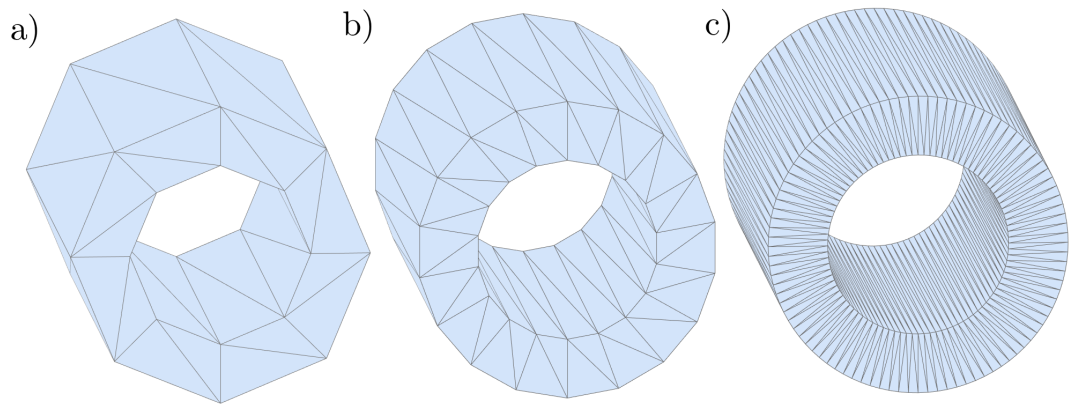


Figure 2.5 *A circular tube triangle mesh model with different level-of-detail: a) 64 triangles b) 144 triangles c) 768 triangles.*

In order to enhance the fidelity of 3D models, 2D bitmaps, such as normal or bump maps is projected to the model surfaces using UV coordinates. A normal map defines the normal at the point of projection and a bump map the relative height from the mapped surface and in the scalar range [16]. By using these maps, small detail is added without having to define the detail using geometry. Current graphics cards feature surface tessellation support which can subdivide given geometry during runtime. With tessellation controlled geometry, detail is added dynamically by using a displacement map as a source of a surface transform [17].

In Vertex CAD software, a commercial ACIS CAD kernel is used for mathematical modeling [18]. The ACIS kernel is also used to convert the mathematical surface model to a polygon mesh. For conversion, the quality and accuracy of the mesh output can be configured. In real-time graphics applications such as games and interactive visualization software, a balance between visual quality and performance has to be taken into account. To attempt to get the best possible quality while maintaining an adequate rendering performance. That causes a problem for the simplification; reduce the number of polygons while maximizing the visual quality. In the next chapter, we review methods of mesh simplification methods in order to reduce geometric complexity.

3. SIMPLIFICATION OF 3D CAD MODEL

In this chapter, we explore ways to reduce the geometry information in order to improve the run-time performance of real-time rendering of 3D CAD models. By reducing the geometry information, we also lose visual information about the object. That means we have to find a balance between required visual quality and simplification amount. Mesh simplification is a well-studied problem, in this chapter, we introduce previous work that proposes techniques to simplify 3D mesh. We especially explore different mesh decimation techniques, such as vertex clustering and edge decimation. Our need is to be able to simplify a wide range of different type CAD models, from small to large mechanical assemblies and from small to large building designs. The ideal simplification method is not specific to the model type, form or features, yet would perform well using any input without excessive per-model based configuration or pre-processing.

Notable mesh simplification surveys include a paper by Arvo et al. where they study different mesh simplification methods for CAD models [3]. The paper excellently collects many research papers for different simplification methods. It also mentions and surveys other mesh simplification techniques, which we leave out, such as remeshing and wavelet decomposition. We specifically focus on techniques that rely on geometric decimation operations, such as edge decimation. As the basis of the decimation operation is simple, the heuristic and algorithms that guide the operations are the most exciting part of the research. Next, in this chapter, we introduce quality evaluation which controls the simplification process, and then we review mesh decimation methods and previous work. Finally, we conclude the review.

3.1 Quality evaluation

Error metrics is necessary to control the simplification process in order to maintain the wanted quality of the simplified mesh. Approximation error defines the deviation between the original and the simplified meshes. The simplification process often uses the approximation error as a guide or terminate the process. A paper by Cignoni et al. describes the following ways to characterize approximation errors [19].

- Locally bounded error, where the error is based on from surrounding faces. Because locality and fast calculation, it is often used in iterative mesh decimation methods.
- Globally bounded error, where the error is known for the whole mesh.
- Other criteria, such as surface boundaries or curvature.
- No approximation error.

For approximation error, different methods for calculating error metrics can be used. For the locally bounded error, a *quadric error metric* (QEM) has been found to be a fast and general approximation. For memory intensive applications, a memoryless approximation is necessary. Memoryless approximation algorithm does not need to save or record the approximation error locally, as it can be calculated on the premise.

Hausdorff distance can be used for high-quality global approximation error. It is also practical for evaluating different simplification methods by geometric quality. Hausdorff distance measures mesh-to-mesh distance [20].

Image-based simplification methods often focus on visual quality. Often mean-square error between images after and before simplification is used as metrics to classify the performance of the simplification method. In view-dependent mesh simplification, viewport location and direction relative to the mesh is often used as an importance factor when calculating the error.

3.2 Vertex clustering

In vertex clustering, a 3D uniform grid is created from polygon mesh. Each vertex is then mapped to the cluster cells. Vertices inside a cluster are merged to one vertex using selected approximation algorithm, such as average, median or least error quadric. Then, the vertices are reconnected between clusters. The first published clustering approach to simplify complex meshes, such as CAD designs, was defined by Rossignac and Borren [21]. Vertex clustering algorithm can process arbitrary polygon inputs, although often the quality of the output is weak and the cluster operation does not attempt to preserve the topology of the mesh. The method has been since extended in many ways. Figure 3.1 illustrates vertex clustering method, where the vertices in a cluster are merged to a single averaged vertex.

In order to handle huge meshes which do not fit main memory, such as medical datasets, Lindstrom proposes an out-of-core method which takes advantage of

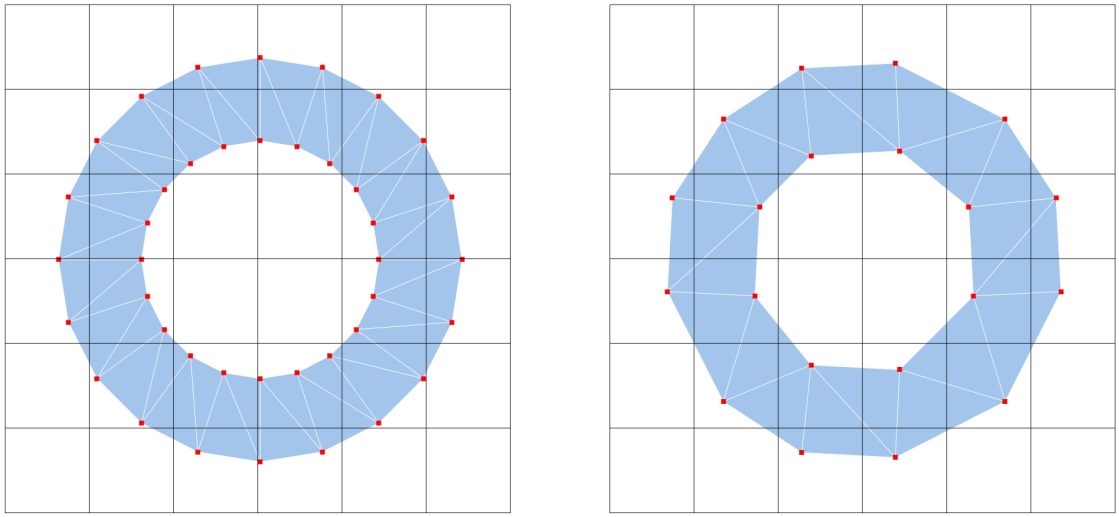


Figure 3.1 *Visualization of vertex clustering, where clusters (left) are merged to a single vertex, which is positioned to the average position of cluster vertices (right).*

quadric error metric [22]. Although vertex clustering is already a fast simplification method, performance improvements have been made. DeCoro and Tatarchuk parallelize the vertex clustering method by modifying the method to use GPU-friendly octree structure and use GPU shaders to implement and execute the simplification [23]. Using the GPU implementation, they were able to simplify the meshes in real-time.

The naive vertex clustering approach does not take account local features, and usually, the resulting mesh has lost defining features such as sharp edges. Many approaches to improve the quality of vertex clustering has been proposed. To improve memory consumption and to preserve more local details, changes to the uniform grid has been proposed. Shaffer and Garland change the uniform grid to binary space partitioned tree [24]. Schaefer and Warren proposed improvement by changing the uniform grid to adaptive octrees [25]. Boubekeur and Alexa introduced geometry aware stochastic vertex selection to preserve geometrical features and topological clustering to preserve topological features [26].

Brodsky and Watson propose a vertex clustering algorithm that starts from coarse approximation [27]. The approximation is then refined by splitting the cluster until the wanted vertex count is reached. Using the same method, Brodsky and Pedersen present a way to partition the model to be able to execute the simplification process on multiple computer clusters [28].

Because of the high performance and generality of the method, it has been used in

run-time simplification and mesh streaming. For example, Limper et al. define a pop-buffer, a polygon model quantization method which they propose to be used as simplification and level-of-detail implementation [29]. Their method is suitable for progressive streaming of large 3D models over a network.

3.3 Vertex decimation

Vertex decimation is a process of selecting, removing vertices and re-triangulating the resulting hole. Similar to all the other decimation methods, the vertex decimation process is iterative and continued until a user-defined constraint is set. Such constraints are, for example, the number of vertices removed and an approximation error threshold. Figure 3.2 shows how a vertex in red is 1) selected and 2) removed. The surrounding geometry is then 3) re-triangulated, reducing triangle count. Six triangles in total are removed in the process.

Vertex decimation is defined by Schroeder et al. where they proposed an iterative multi-pass vertex removal [30]. A vertex is iteratively selected for removal, and all the connected faces are removed. Local triangulation process is used to patch the hole caused by the vertex removal.

Klein et al. further improves the mesh quality of the vertex decimation by using Hausdorff distance between the original and the simplified mesh as an approximation error metric [20]. The minimum Hausdorff distance is user-defined and used as a constraint to terminate the decimation process as it is about to be exceeded. Multiresolution surface modeling technique has been proposed by Soucy and Laurendeau to improve the triangulation process by optimizing the equiangularity of the simplified mesh surface [31]. Chuon and Guha proposed a volume cost based method [32]. Their method improves the mesh quality by classifying a vertex as hyperbolic or non-hyperbolic and calculating the cost based on the spherical volume that is lost if the vertex is removed.

3.4 Edge decimation

In edge decimation or edge collapse, an edge that connects two vertices is removed from the mesh and replaced with a single vertex. Different edge decimation methods have proposed various ways to place the new vertex to minimize the approximation error. The operation is illustrated in Figure 3.3, the edge in red is 1) selected and then 2) collapsed. As the figure shows, two triangles are removed after the operation is completed.

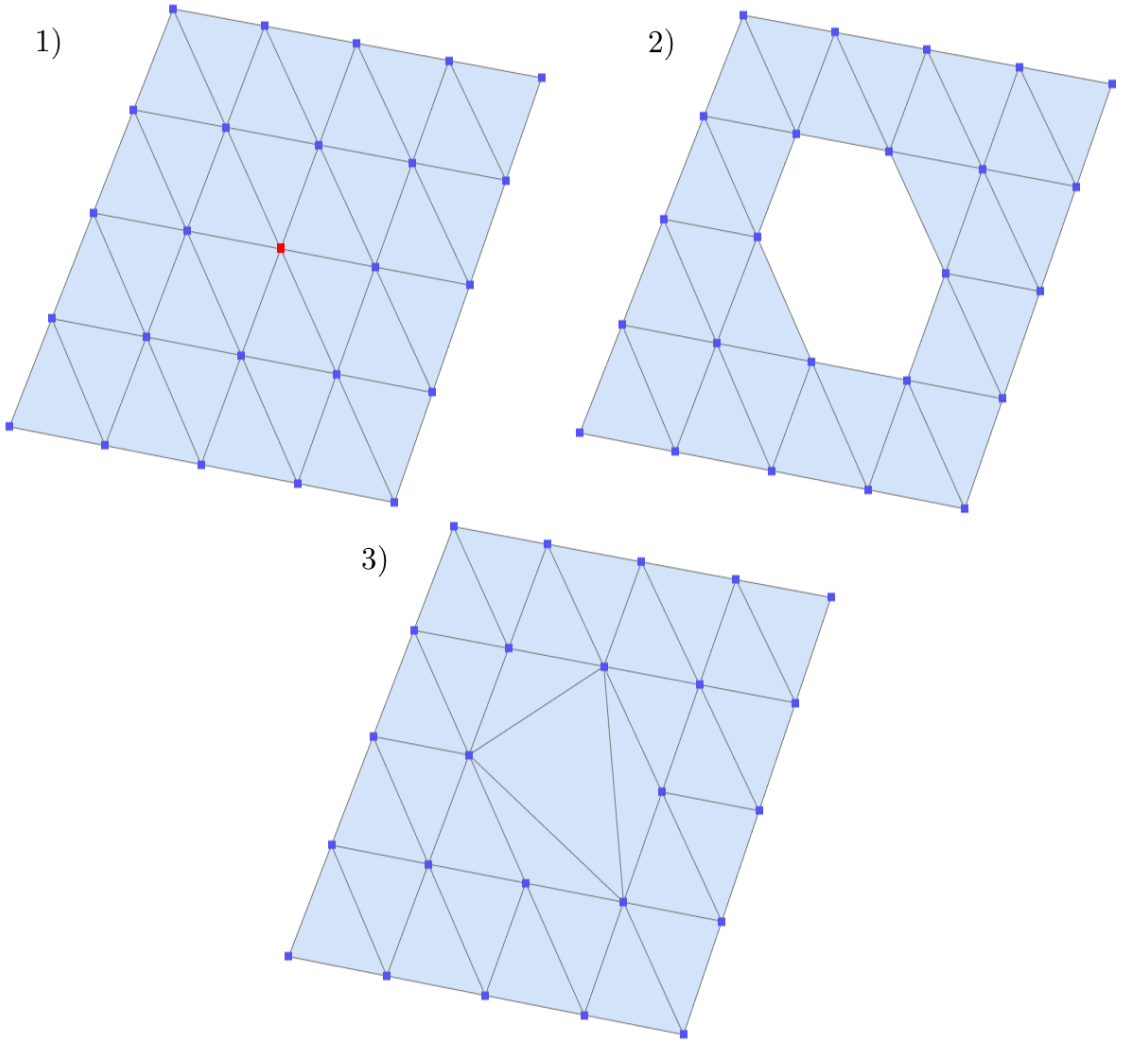


Figure 3.2 *Visualization of vertex decimation. The vertex in red is 1) selected and 2) removed. Finally, 3) the resulting hole is patched.*

Hoppe et al. samples random points along the original mesh, and uses them to compute energy metric to minimize volume error and guide the edge collapse, edge swap, and edge split operations [33]. Next, Hoppe proposes that only edge collapse operation is needed for efficient simplification, and proposes a mesh representation that can be transformed in a level-of-detail domain by edge collapse and edge split operations [34]. Hoppe also extends the progressive mesh optimization method for view-dependent simplification by adding a screen-space geometric error [35]. Franc and Skala adapt the vertex decimation method by Schroeder et al. and use edge decimation instead in order to speed up the simplification process [36].

Roufard and Rossignac propose a method to calculate local geometric error by defining, for every vertex, planes of each triangle attached to it and compute the error as a sum of squared distances from the planes [37]. Garland and Heckbert simplify the

method proposed by Roufard and Rossignac by showing that the same error metric can be defined using quadric error metric [38]. In their next paper, Garland and Heckbert extend the method to take account vertex attributes, such as color, normal and texture coordinates [39]. Quadric error metric based decimation remains a fast and consistent method. Bahirat et al. derive their version of QEM based decimation algorithm by taking account boundary preservation and focus on optimizing the simplification method for virtual reality applications [40].

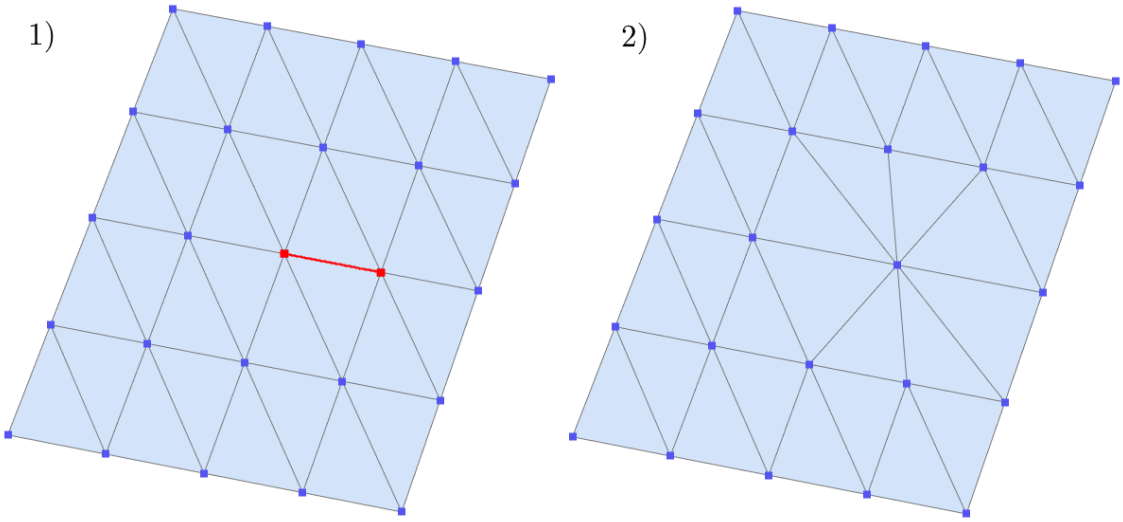


Figure 3.3 *Visualization of edge decimation. The edge in red is 1) selected and 2) collapsed.*

Lindstrom and Turk build on the quadric error metric and also use volume and boundary constraints to perform memoryless simplification suitable for memory constrained applications [41]. Hoppe builds on the work of Garland and Heckbert, and proposes an improved version of the quadric error metric by adding vertex attributes, volume preservation, and memoryless simplification [42]. Salinas et al. propose a pre-processing step where planar proxies are detected and used to guide the decimation [43]. Their method requires significant pre-processing time and produces an improved result when simplifying models with planar surfaces.

Parallelized edge decimation methods have been proposed to take advantage of programmable GPUs. Papageorgiou and Platis introduce GPU based edge decimation method. They divide the mesh into areas that are simplified independently of each other [44]. Hu et al. parallelize the view-dependent progressive meshes by Hoppe using GPU programmable pipelines [45, 46]. Odaker et al. focus on view-dependent real-time simplification describe a half-edge collapse method that is run in parallel on the GPU [47]. They achieve fast simplification time although they lose quality compared to other iterative approaches. Lee and Kyung also introduce a similar

GPU accelerated edge collapse method and use a more sophisticated way to update the mesh to improve the simplification quality [48].

3.5 Triangle decimation

Similar to vertex and edge decimation, some methods have used triangle decimation or triangle collapse. Decimating a triangle is the same as performing three edge collapses, which leads to the removal of four triangles. Figure 3.4 illustrated the process of triangle collapse, where a triangle is 1) selected and 2) collapsed.

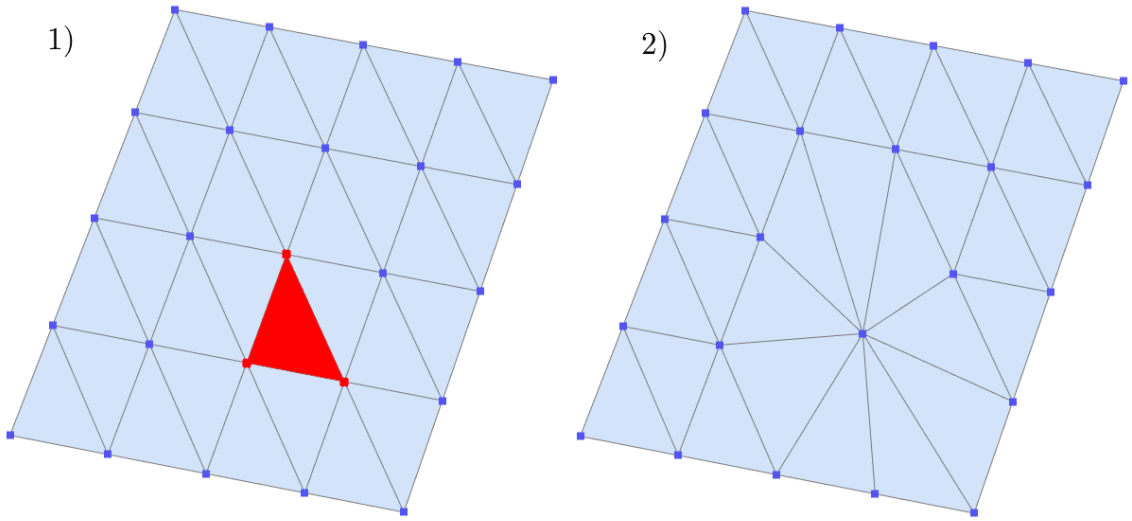


Figure 3.4 *Visualization of triangle decimation. The triangle in red is 1) selected and 2) collapsed.*

Haman proposes triangle decimation and computes a weight value from curvature at triangle vertices [49]. The triangles are then inserted to a removal priority queue based on the weight value. Triangles in planar surfaces have low weight and are first to be decimated. Gieng et al. builds on the triangle decimation method and proposes adding a triangle area term for weighting calculation [50]. They also use past information of triangle collapse operations when calculating triangle weights. Gieng et al. expand the method by proposing a triangle collapse operation to produce a continuous level-of-detail variation of the mesh [51]. Isler et al. propose a hybrid method where they use triangle decimation for flat areas and edge decimation for feature defining areas [52]. They classify vertices into three different categories, flat, edge, and feature. Visual importance of a triangle is based on the classification of its vertices.

3.6 Image-based decimation

Image-based methods use data gathered from images of rendered mesh as an error metric. The method if often computationally heavy as the mesh has to be rendered multiple times during the mesh simplification. For CAD models, image-based decimation is often beneficial in cases where the inside geometry is unimportant for visualization. In some visualization purposes, inside geometry is essential and must be included in the final output. The quality of the results depends on the count and locations of the viewpoints, and thus manual viewpoint placement is beneficial for simplification quality.

Lindstrom and Turk propose calculating edge collapse cost from luminance based on the mean-square error between images before and after edge collapse [53]. They render the mesh from multiple uniformly distributed viewpoints. Lindstrom and Turk mention that image-based decimation might yield better results for applications where visual importance is more valuable than geometric accuracy when comparing to the original model. Their results show that quadric error metric based edge decimation is a much faster method than their proposed image-based method. Zhang and Turk pre-compute a view-independent visibility measure for each triangle and combine the information with quadric error metric based edge decimation [54]. Using the visibility measure, they were able to produce visually better results when compared to QEM-based edge decimation. Lee et al. define mesh saliency, a measure of regional importance, inspired by human visual system cues. They use pre-computed saliency to guide the quadric error metric based decimation method [55]. Their simplification results show improvement in preserving details such as eyes and nose better. Qu and Meyer derive view-dependent simplification algorithm based on models of the human visual system [56]. They generate an importance map and use it to compute the importance value for each vertex to be used in the decimation algorithm.

3.7 Other decimation methods

Cohen et al. propose appearance-preserving simplification, where in addition to generating an approximation mesh, create texture and normal maps based on the original mesh surfaces [57]. From the results, it is seen that the normal maps help to preserve the high-fidelity of the original mesh. Alvarez et al. propose a simplification method that uses the Growing Neural Gas algorithm to generate a simplified set of vertices from the input mesh and then reconstruct faces using the generated set of vertices [58]. Gonzales et al. propose user-assisted simplification by giving the user

control to decide each sub-mesh simplification amount [59]. They use information from CAD to divide the mesh to sub-meshes. Gao et al. use feature recognition and suppression for CAD mesh simplification [60]. Their method removes small features from the mesh while preserving the overall form under heavy simplification when compared to QEM-based edge decimation. Kanai et al. propose three different methods of removing invisible CAD model parts and features [61]. They use rendering to determine if a part or feature is visible.

3.8 Summary

We reviewed various mesh decimation methods. In short, we found three mesh decimation operations and vertex clustering to simplify a polygon model. In vertex decimation, a vertex is removed, and the resulting hole is patched using a method specific to the application. In edge decimation, an edge is collapsed to a computed or selected collapse point. In triangle decimation, a triangle is collapsed, it is similar to collapse three edges at once. Lastly, in vertex clustering, the vertices are assigned to grids in 3D space. The vertices inside a grid are then merged to a single vertex.

Vasts amount of methods exists to select which vertex, edge, or triangle should be decimated. Usually, the decision is based on an error metric, which is classified either local geometric error or global geometric error. Global error is more costly to compute when comparing to local error. A local error also produces results that are close to methods using global error [38]. One of such local metric is quadric error metric which has been used and extended by many [38, 39, 40, 41, 42]. We also reviewed image-based decimation methods, where images or visual cues are used to determine the vertex error weights [53, 54, 55, 56]. In view-dependent simplification methods, the location of the viewpoint is used to guide the decimation [45, 46, 47]. Parallelization is used to accelerate the decimation process, and some parallelized decimation techniques are designed to run real-time [44, 45, 46, 47, 48].

Based on the review, we selected edge decimation and quadric error metric introduced by Garland and Heckbert as a basis for our mesh simplification process [38]. The reasoning is that the quadric error metric has been extended by many and various implementations use quadric error metric. Furthermore, the original method is easy to implement and extend. Also, the performance of the original method is adequate for our offline simplification process. In the next chapter, we cover the implementation of a mesh export process and mesh simplification which is based on the quadric error metric.

4. IMPLEMENTATION

In this chapter, we describe a process to simplify the 3D CAD scene from Vertex CAD software. We cover the process of exporting the CAD model as a triangle mesh and the simplification method implemented. We select a simplification method based on the quadric error metric as proposed by Garland and Heckbert based on the simplicity and popularity of the method [38]. We propose an edge decimation cost threshold in order to handle the meshes of different complexities gracefully. We also extract metadata, such as names, dimensions, descriptions from the model which is attached to the final exported result. The resulted triangle mesh, materials, and metadata is exported to file in the glTF file format [62].

The implementation is programmed in C++ language, and uses the Microsoft STL implementation of the C++ standard library. We also use Microsoft glTF library for glTF exporting [63]. C++ language was selected as the Vertex CAD is build using C++. At the time when the implementation began Microsoft's glTF library was only available C++ library that supported glTF 2.0 format. glTF is a file format that is specified by Khronos organization, which is well known in the graphics industry from OpenGL and Vulkan specifications. We selected the glTF file format because of the open nature of the format and it has been gaining lot of support from various organizations such as Microsoft and Mozilla. Because the glTF is a relatively new format at the time of writing, a negative aspect of glTF format is that it is 2.0 specification support may be lacking across software.

4.1 Overview of the process

At the beginning of the exporting process, we have a 3D CAD scene S. The scene S contains a set of geometric objects $\boldsymbol{M} = \{m_0, m_1, ...m_i\}$. We start the export process for a scene S which is explained in Program 4.1.

```
1 function export scene S:
2   scene triangle mesh: T
3   foreach geometric object M in S:
4     T_m = triangulate object M
5     Ts_m = simplify T_m
6     append mesh Ts_m to T
7   append T to glTF file
```

Program 4.1 *Overview of the CAD scene export process.*

Each geometric object m_i in the scene S is iteratively processed. A triangular mesh T_m is extracted from the geometric object m_i, and then the mesh is simplified to Ts_m. The resulting mesh is appended to the final triangular mesh T. Metadata is also collected from the geometric object during the iteration. After the iteration of the geometric object has been completed, we have a triangle mesh T that contains the CAD scene as a simplified mesh. In our implementation, the triangle mesh is split by materials, which are found from the CAD model surfaces, as each material is rendered separately during run-time. Finally, the triangle mesh, materials, and metadata are saved to a file in a glTF format.

Internally, the CAD scene or model is built from either mathematical models or polygonal visual models. A mathematical model allows for precise modeling of three-dimensional objects and scenes, and visual polygonal models are used for visualization purposes. In Vertex CAD software, the scene can contain both models that are mathematically represented or have polygonal mesh representation. If the model is mathematically represented, it is converted to polygonal representation for visualization purposes during run-time. The conversion is done by the ACIS CAD kernel, which Vertex CAD uses. ACIS is a modeling kernel library which is a commercially produced software [18]. Furthermore, the ACIS features a faceting component that produces a triangle mesh presentation of the mathematical model. This faceting is used in Vertex CAD to create the visualization model.

Typically, Vertex CAD models and scenes are described as a tree hierarchy. For example, a mechanical model is represented by a top-level assembly that further contains child assemblies and parts. In other words, an assembly is a collection of parts or child assemblies. Each part object contains geometric data. In our implementation, this means that every part of the model is processed and simplified separately from others. Thus the whole CAD scene or model is not simplified as one large unit.

4.2 Model data representation

The exporting algorithm iterates through objects in the CAD scene which contain geometric data. In an iteration, a data structure object T_m representing the geometric object specific for the exporting process is built from the geometric data. The vertices and indices of the mesh generated by ACIS or which are already present in the geometric data are saved to the data structure. We split the triangle mesh per material and use a unique material index to map the corresponding triangle mesh for that material. Furthermore, a transformation is saved along with the vertices and indices. The transformation matrix is a 3 by 4 matrix which represents the geometry's absolute scene space position and rotation in double-precision floating-point format. Double-precision is selected as Vertex CAD internally uses double-precision for point transformation. The data structure of a geometric object is described in Program 4.2.

```
1 structure geometric object {
2   map of integer, triangle mesh: mesh by material
3   matrix3x4: transform
4 }
```

Program 4.2 *Program structure of a geometric object.*

The triangle mesh that is specific to a unique material index is built from vertices and indices. A vertex has a position, normal and texture coordinate attributes. Position and normal are represented using three-dimensional single-precision float-vector and texture coordinate with two-dimensional single-precision float-vector. Each attribute is in a separate list and indexed using the mesh indices. The data structure of a triangle mesh is described in Program 4.3.

```
1 structure triangle mesh {
2   list of vector of 3 float: positions
3   list of vector of 3 float: normals
4   list of vector of 2 float: texture coordinates
5   list of integer: indices
6 }
```

Program 4.3 *Program structure of a triangle mesh.*

The position and normal coordinate is in mesh local space. The transformation from the geometric object is used to get the world space location of the mesh. Normal is a unit vector between which defines a perpendicular vector the surface point. Texture coordinate vector is from scale $]0, 1[$ is used to map the texture of the material at the surface point.

4.3 Data structure for mesh decimation

In order to decimate edges from the mesh, we need to have information about neighboring vertices and triangles. The input triangle mesh data structure does not contain neighbor information which makes it very slow to find neighbors purely from it. To accelerate the simplification process, we designed data structures that contain information about neighbors and relations.

4.3.1 Definition

We define three primitives vertex, edge, and triangle. Each primitive is defined with its own data structure and has connections to other primitives. A vertex is defined as a position, and it has a list of triangles and edge indices to determine which edges and triangles it is connected to. The vertex structure is shown in Program 4.4.

```
1 structure vertex {
2   vector of 3 float: position
3   list of integer: edges
4   list of integer: triangles
5 }
```

Program 4.4 *Program structure of a vertex.*

An edge is constructed from a begin and end vertex index, and a list of triangle indices the edge is connected to. List of triangle indices in non-manifold cases can be more than two or in boundary case only one. Also, implementation specific properties are included such as an iterator to the cost ordered map and a collapse target if the edge is to be collapsed. Edge structure is illustrated in Program 4.5.

```
1 structure edge {
2   integer: vertex begin
3   integer: vertex end
4   list of integer: triangles
5   vector of 3 float: collapse target
6   heap iterator:  it
7 }
```

Program 4.5 *Program structure of an edge.*

Finally, the triangle contains information about the surface, such as material index, vertex and texture coordinate indices, a normal vector and a state if the triangle has been removed. The triangle structure is the most critical structure when the

mesh is converted back to a triangle mesh. Program 4.6 illustrates how the triangle structure is laid out in the implementation.

```
1 structure triangle {
2   integer: material
3   vector of 3 integer: vertices
4   vector of 2 float: texture coordinates
5   vector of 3 float: normal
6   boolean: is removed
7 }
```

Program 4.6 *Program structure of a triangle.*

As an implementation specific detail, none of the structures does own the structure it is referencing to. The referencing is done by indexing to a list which owns the data structure. Figure 4.1 visualizes the relations between vertex, edge and triangle structures.

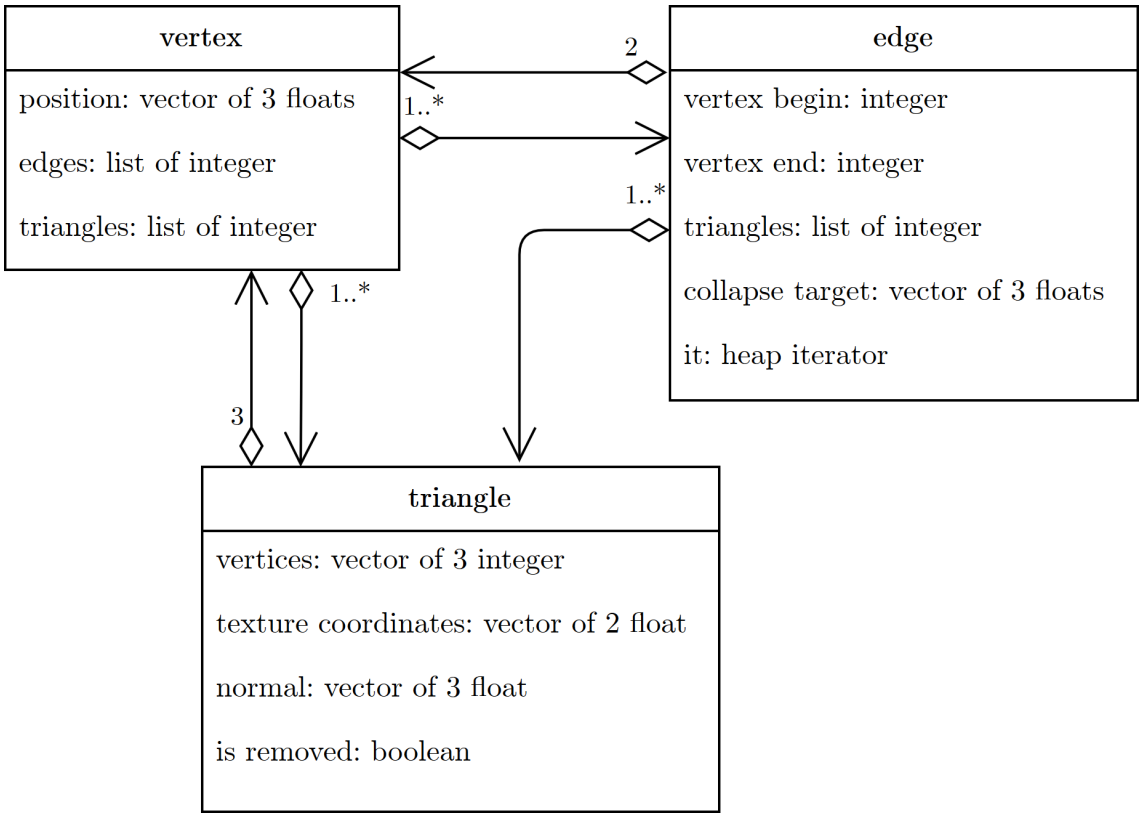


Figure 4.1 *Visualization of custom decimation data structure relations.*

4.3.2 Construction

To construct the data structures, we start by iterating the triangles in the triangle mesh. Note that the triangle mesh is divided per material. If the model has more than one material, we can be almost sure that overlapping vertices exists. We assume that the input has only triangle surfaces and are ordered by indices. We start by building vertices of the triangle T. For each vertex of the mesh triangle we attempt to find if some vertex is located at a position already defined for an existing vertex. We do this by using spatial hashing with a multi hash map, which can contain one or more elements for a single hash value. The multi hash map is specific to a single material. It ensures that vertices that are at the same position but are part of different material, do not get mixed. By not mixing the vertices with vertices of different material, we create a boundary vertex. In conjunction with boundary constraints, we do not collapse the edges on material boundaries. A vertex is created if one was not found at the position. Finally, we have vertices $V = \{v_i, v_j, v_k\}$. Program 4.7 illustrates the algorithm pseudocode.

```
1  function find or add vertex at P:
2    integer64: position hash = get hash for P
3    list of hash entries: P hash entries
4    P hash entries = find position hash in vertices multi hash map
5
6    for hash entry in P hash entries:
7      vertex: V = hash entry value
8      if absolute distance V position and P < epsilon:
9        // Vertex was found by spatial hashing -> return
10       return V index
11
12   // Create new, as vertex was not found by spatial hashing
13   create vertex V with position P
14   add V to vertices multi hash map with position hash
15   return vertex V index
```

Program 4.7 *Algorithm to find or add a vertex with the given 3D position vector.*

To calculate the hash value, we use optimized spatial hashing function proposed by Teschner et al. [64]. They define a cell size which they use to discretize the 3D position. The discretization is done by dividing the position by the cell size and truncating the result. Similar to vertex clustering, we now have a 64-bit integer vector that defines the location of the vertex's cluster. Finally, they use a hash function for the discretized 3D position to calculate the hash value. The hash function implementation is defined in Program 4.8.

```
function get hash for position P:
  double: C = 1 / 0.00001 // inverse of cell size in meters
  array of integer64: [x, y, z] = truncate (C * P)
  // Prime number defined by Teschner et al.
  array of integer: [p1, p2, p3] = [73856093, 19349663, 83492791]

  // Hash function from Teschner et al.
  integer64: hash = (x * p1) xor (y * p2) xor (z * p3)
  return hash
```

Program 4.8 *Algorithm to compute a 64-bit integer hash hash for 3D position vector.*

Triangle contains three edges, which are connected between vertices $\{v_i, v_j\}$, $\{v_j, v_k\}$, and $\{v_k, v_i\}$. As triangles in most cases share edges, we attempt to find if an edge already exists with given the vertex indices $\{i_v, j_v\}$, $\{j_v, k_v\}$, and $\{k_v, i_v\}$. If such edge does not exist, an edge is created. The algorithm is defined in Program 4.9.

```
function find or add edge for {i_v, j_v}:
  integer64: e hash = get hash for {i_v, j_v}
  if edge e with e hash in edge hash map:
    return edge e index
  else
    create edge e with with vertex indices {i_v, j_v}
    add e to edge hash map with e_hash
    return edge e index
```

Program 4.9 *Algorithm to find or add an edge for given endpoint vertices.*

We use a hash map with 64-bit integer keys to accelerate the search. The vertex indices are 32-bit unsigned integers. To create the key, we combine to one 64-bit unsigned integer. We also order the indices by magnitude to make the hash function order-independent as the edges in our data structure do not have direction. The hash function is defined in Program 4.10.

```
function get hash for edge {i_v, j_v}:
  if i_v > j_v:
    swap i_v and j_v
  // bitwise shift left 32 bits and bitwise or
  integer64: hash =  i_v << 32 | j_v
  return hash
```

Program 4.10 *Algorithm to compute a 64-bit integer hash for an edge.*

After finding vertices and edges, we associate a triangle T with vertices V and vice versa. Program 4.11 describes the creation of the data structures in detail. First,

in lines 4-12, we find vertices and edges as described previously. In lines 14-19 a triangle is created, for the triangle, we also calculate a normal vector and add material identifier and texture coordinates. Finally, in lines 22-23, we associate vertices V and edges E with triangle T. In lines 25-27, edges E with vertices V.

```
1  function create data structures for triangle mesh T_m:
2    for mesh per material M in T_m:
3      for mesh triangle M_t in M:
4        // Find or add vertices, return vertex indices
5        integer: i_v = find or add vertex in M_t vertices at index 0
6        integer: j_v = find or add vertex in M_t vertices at index 1
7        integer: k_v = find or add vertex in M_t vertices at index 2
8
9        // Find or add edges, return edge indices
10       integer: i_e = find or add edge {i_v, j_v}
11       integer: j_e = find or add edge {j_v, k_v}
12       integer: k_e = find or add edge {k_v, i_v}
13
14       integer: i_t = get next triangle index
15       triangle: T = create triangle at index i_t
16       set T vertices = {i_v, j_v, k_v} // set triangle vertices
17       set T normal = calculate normal from vertices at {i_v, j_v, k_v}
18       set T texture coordinates = get M_t texture coordinates
19       set T material = get M_t material
20
21       // Associate edges and vertices with triangle
22       add triangle index i_t for edges at {i_e, j_e, k_e}
23       add triangle index i_t for vertices at {i_v, j_v, k_v}
24       // Associate vertices with edges
25       add edge index i_e for vertices at {i_v, j_v}
26       add edge index j_e for vertices at {j_v, k_v}
27       add edge index k_e for vertices at {k_v, i_v}
```

Program 4.11 *Algorithm to create data structures for the edge decimation process.*

4.4 Simplification

The resulting mesh of the geometric object T_m is then simplified. In the end, we selected the quadric error metric based edge decimation method by Garland and Heckbert. The decision based on the available research and high appraisal of the quality, simplicity, and performance of the method. Another method that was closely evaluated was the method by Lindstrom and Turk [41]. Based on the available research, performance was much worse than the QED based method, and simplification quality gain was minimal. As the export is done offline and not during the critical

rendering process, real-time focused simplification methods were discarded. Furthermore, viewpoint-dependent and image-based methods add significant performance overhead and mostly are not suitable for general case CAD simplification.

The simplification process begins by building our neighbor look-up friendly data structures which largely simplifies and speeds up the edge decimation operation. Then, we proceed by implementing the algorithm as described by Garland and Heckbert. We calculate the vertex errors, compute initial edge collapse costs and start iterative edge decimation. We use an edge decimation cost threshold along with maximum face decimation amount as conditions to terminate the iteration. Finally, we rebuild the triangle mesh from the data structures and return the simplified mesh. In our implementation, the quadric error metric based edge decimation is implemented as described in Program 4.12.

```
1 function simplify geometric object G:
2   create data structures for G
3   compute quadrics
4   compute edge collapse costs and targets
5   do iterative edge decimation
6   Gd = rebuild triangle mesh
7   return Gd
```

Program 4.12 *Overview of the process to simplify a geometric object.*

4.4.1 Edge decimation cost threshold

In order to retain the output quality stable regardless of the input mesh complexity, we propose an edge decimation cost threshold T_c. As different models have different levels of geometric complexity, a face target count does not work alone to terminate the decimation process in most situations. For example, a building design, along with topologically complex items, often have a lot of box-shaped geometries such as walls. Such walls often are topologically simple items and usually cannot be decimated without a significant loss in visual and geometric quality. The edge decimation cost threshold is derived from the model's bounding box scaled by user-defined value: $T_c = diag(T_{m_{bb}})C_u$. Where C_u is a user configurable input variable, giving a model complexity independent way to control the output quality.

4.4.2 Garland and Heckbert edge decimation algorithm

Next, we start the simplification process. The algorithm summary is well described in Garland and Heckbert paper in section 4.1 [38]. First, we calculate the symmetric

4x4 matrices Q for each vertex. The matrix Q is used to characterize the geometric error at the vertex. Then, we select all valid edges for decimation, compute optimal contraction target v_c and error for each valid edge (v_1, v_2) and place the edges to a heap with the minimum cost edge at the top. Finally, we iteratively remove the edges of the least cost from the heap, contract the edge and update the cost of edges involving the edge until the target cost threshold is reached.

4.4.3 Computing Q for vertices

In order to calculate the symmetric matrix Q for given vertex v, we calculate the planes $\boldsymbol{P} = [\, \boldsymbol{p_1}\ \boldsymbol{p_2}\ ..\ \boldsymbol{p_n}\,]$ of every triangle. Where $\boldsymbol{p_i} = [\, a\ b\ c\ d\,]^T$ represents a plane for equation $ax + by + cz + d = 0$. From the plane $\boldsymbol{p}_i$, we a create symmetric matrix $\boldsymbol{K}_p$ in the form:

$$\boldsymbol{K}_p = \boldsymbol{p}\boldsymbol{p}^T = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}.$$

For every vertex of a triangle, we sum the matrix $\boldsymbol{K}_p$ to the matrix $\boldsymbol{Q}$ belonging to the vertex. The procedure is also explained in Chapter 5 of Garland and Heckbert paper [38]. The pseudocode for our implementation of the algorithm is defined in Program 4.13.

```
1 function compute quadrics:
2   for every triangle T in triangles:
3     float: D = - dot product of T normal and V position
4     plane: p = { T normal, D }
5
6     symmetric matrix: K_p = create symmetric matrix from plane p
7
8     for every vertex V in triangle T:
9       sum K_p to symmetric matrix Q of vertex V
```

Program 4.13 *Algorithm to calculate the matrix Q for vertices.*

4.4.4 Computing edge collapse costs and targets

Next, we iterate through the edges. We check if the edge is valid, compute initial collapse cost and target for it and finally add it to the heap using the collapse cost

as a key. In our case, an edge is valid if it has two triangles attached to it. Thus boundary edges or edges with more than two triangles are left out from the collapse operation. The algorithm for computing the costs and target is defined in Program 4.14.

```
1 function compute edge collapse costs and targets:
2   for every edge E in edges:
3     update collapse cost and target for edge E
4
5     if edge E triangle count == 2:
6       add edge E to heap tree using E collapse cost
```

Program 4.14 *Algorithm to compute the collapse cost and the collapse target for edges.*

The procedure for calculating the cost of collapse and finding a target for edge collapse is explained in Chapter 4 of Garland and Heckbert paper [38]. Error function for given vertex $\boldsymbol{v} = [v_x v_y v_z 1]^T$ is defined as $\Delta(\boldsymbol{v}) = \boldsymbol{v}^T \boldsymbol{Q} \boldsymbol{v}$. The error function is finally transformed to form:

$$\boldsymbol{v}^T \boldsymbol{Q} \boldsymbol{v} = q_{11}x^2 + 2q_{12}xy + 2q_{13}xz + 2q_{14}x + q_{22}y^2 + 2q_{23}yz + 2q_{24}y + q_{33}z2 + 2q_{34}z + q_{44}.$$

In order to calculate a cost for edge collapse $(\boldsymbol{v}_1, \boldsymbol{v}_2) \rightarrow \boldsymbol{v}_c$, the quadrics of the vertices are summed $Q_c = Q_1 + Q_2$ and v_c is found that minimizes the error function $\Delta(v_c)$. To find the $\boldsymbol{v}_c$, the equation:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{v}_c = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

is to be solved for $\boldsymbol{v}_c$. If the matrix $\boldsymbol{Q}_c$ is invertible, the $\boldsymbol{v}_c$ is solved from:

$$\boldsymbol{v}_c = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{12} & q_{22} & q_{23} & q_{24} \\ q_{13} & q_{23} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Otherwise, the $\boldsymbol{v}_c$ is either attempted to be found from along the segment $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ or by selecting one of the segment endpoints or midpoint.

In our implementation, we follow Garland and Heckbert closely. The implementation

pseudocode for updating the edge collapse cost and the collapse target is presented in Program 4.15. In lines 8-10, the case when matrix $\boldsymbol{Q}_c$ is invertible is handled. Otherwise, lines 11-23 handle the opposite. In contrast to Garland and Heckbert, to save time in the case when the matrix is not invertible, we iterate the segment with discrete steps and select a target point of the smallest cost.

```
1  function update collapse cost and target for edge E:
2    vector of 3 float: Target
3    float: Cost
4
5    // Qc = Q1 + Q2
6    symmetric matrix: Qc = sum of edge vertices Q matrices
7
8    if Qc is invertible:
9      Target = get [x y z]  from inverse of Qc * vector [0 0 0 1]
10     Cost = compute cost for Target
11   else:
12     vector of 3 float: Iterate Vector
13     Iterate Vector = calculate iteration vector (E v2, E v1)
14     integer: Iterations = 10
15
16     iterate integer I range (0, Iterations):
17       vector of 3 float: Position
18       Position = E V1 position + Iterate Vector * I
19       This Cost = compute cost for Target
20
21       if This Cost < Cost:
22         Target = Position
23         Cost = This Cost
24
25   set E collapse target to Target
26   set E collapse cost to Cost
```

Program 4.15 *Algorithm to update the collapse cost and the collapse target for an edge*

4.4.5 Decimating the edges

We now have a heap which we begin to iterate and process the edges. In our implementation there are three conditions which are used to terminate the iteration:

1. The edge heap becomes empty. I.e. all edges that were marked for removal were processed.

2. The number of triangles removed is equal to or greater than the specified target number of triangles to be removed.

3. The cost of the edge removal is equal or greater than the specified edge decimation cost threshold T_c.

The edge decimation cost threshold is determined by scaling the length of the model's bounding box diagonal by a user-defined scalar value. The length of the model's bounding box diagonal ensures that the cost threshold does not need to be modified between different models which differ in size and complexity.

The collapse algorithm is shown in Program 4.16. In lines 3-8, we begin the iteration of the edge heap and check against the two termination conditions before decimating the edge. In lines 14-17, we check if some of the resulting triangles after the edge E collapse would be flipped, the edge E is connected to a border, or the edge E collapse would result in a non-manifold surface. If one or more of the conditions are true, then the collapse operation is invalid. In lines 20-28, we create new a vertex V_n at the edge E collapse target, mark triangles attached to the edge E as removed, update edges and triangles of the edge E with the vertex V_n, set the matrix Q and compute neighbors edge collapse costs for the vertex V_n. The process is continued until a termination condition is reached, or all edges in the edge heap are processed.

```
1  function do iterative edge decimation :
2    integer: number of triangles removed = 0
3    while edge heap is not empty:
4      pop edge E from edge heap
5      // Check for stop conditions (face count removed and Tc)
6      if number of triangles removed >= triangle remove count target
7         or E collapse cost > edge decimation cost threshold:
8        break
9
10     // Skip collapse of this edge
11     // if resulting triangle(s) would be flipped
12     // if edge is connected to a border edge
13     // or edge collapse will cause non-manifold surface
14     if does neighbor triangle flip after collapse for edge E
15        or is edge E connected to border
16        or will cause non manifold surface after collapse edge E:
17       continue
18
19     // Create new vertex at collapse target of edge E
20     vertex: Vn = create vertex at E collapse target
21
22     for triangle T in edge E:
23       mark T as removed
24       number of triangles removed += 1
25
26     update edges and triangles of edge E with new vertex Vn
27     merge similar edges in vertex Vn
28     set Q and compute edge collapse costs for vertex Vn
```

Program 4.16 *Algorithm to collapse edges.*

4.4.6 Deciding if collapse operation is valid for an edge

Before the edge is collapsed, there are three tests performed which ensure that the collapse operation is valid. First, we test if the edge collapse will flip any triangle that is connected to the edge vertices (v_1, v_2). As Figure 4.2 illustrates, triangle flip causes intersecting geometry which will produce unwanted artifacts for the mesh.

Program 4.17 shows how every triangle connected to the edges by vertices are tested. In lines 6-7, the triangles containing the collapse candidate are excluded because they will be collapsed. In lines 9-10, a new triangle ΔT which represents the result after decimation is created. In lines 12-17, a normal for the ΔT is computed, and it is tested against the normal of the current triangle. Given the properties of the dot product, if the dot product is negative, we know that the angle between the normal

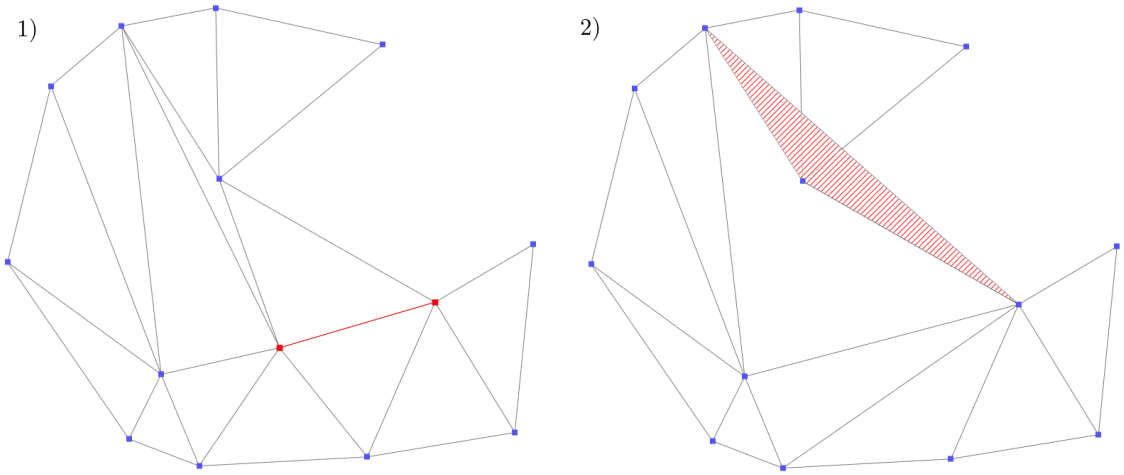


Figure 4.2 *Visualization of 1) invalid edge collapse operation, 2) where the triangle in red stripes is flipped after the collapse of the red edge.*

vectors is greater than 90 degrees. If the dot product is close to minus one, then the triangle after edge collapse is flipped completely. Thus the new normal would be almost perpendicular to the opposite direction, making the decimation operation invalid.

```
1  function does neighbor triangle flip after collapse for edge E:
2    for vertex V in E:
3      // Loop through triangles connected to vertex
4      for triangle T in vertex V:
5        // Edge which is part of the triangle will be collapsed always
6        if edge E in triangle T:
7          continue
8
9        triangle: delta T = create triangle from T
10       in delta T replace V position with E collapse target
11
12       vector of 3 floats: normal = calculate normal for delta T
13       // Check if normal is almost perpendicular
14       // to the opposite direction
15       float: d = dot product of T normal and delta T normal
16       if absolute (d) > 1 - Epsilon:
17         return true
18
19   // Function completed checking triangles validity successfully
20   return false
```

Program 4.17 *Algorithm to test if a triangle will flip when an edge is collapsed.*

To prevent possible discontinues on model borders, the second test ensures that

none of the edge vertices lie on a border edge. Program 4.18 illustrates an algorithm that iterates all edges connected to the edge vertices and tests if one of them is on a border.

```
1 function is edge E connected to border:
2   for every vertex V in E:
3     // Check if the edges connected to the edge E vertices
4     // are part of boundary or non-manifold mesh
5     for every edge Ev in V:
6       if Ev triangle count != 2:
7         return true
```

Program 4.18 *Algorithm to test if an edge is connected to a border.*

The third test, analyses if the edge collapse results in two edge merges. This way we ensure that the edge collapse does not result in non-manifold geometry. Figure 4.3 illustrates a case where an edge collapse results in non-manifold geometry. The figure shows four edges that are merged. In a valid edge collapse operation, two edges should be merged.

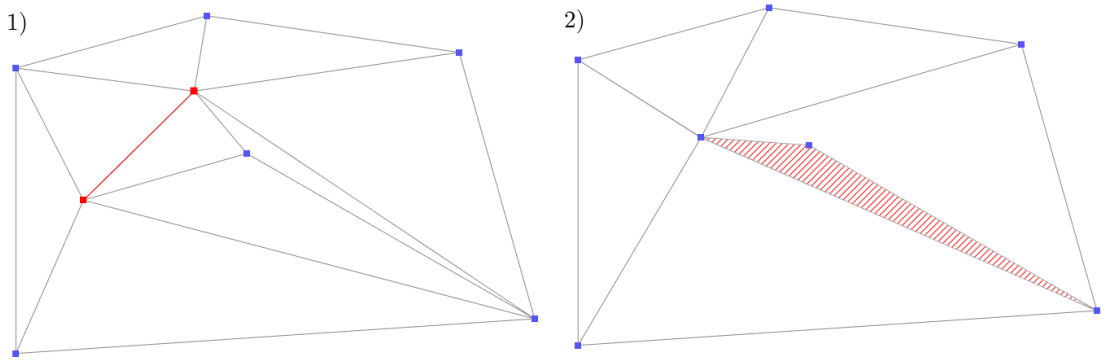


Figure 4.3 *Visualization of 1) invalid edge collapse operation, 2) where the triangle in red stripes is non-manifold after the collapse of the red edge.*

Program 4.19 tests if there are two edge merges when the edge E is collapsed. Every triangle connected to the edge vertices is tested if it resulted in an edge that would exist after the collapse. In a valid case, there should be two pairs of edges that are merged when an edge is collapsed.

```
1  function will cause non manifold surface if collapse edge E:
2    hash set: Edge Hashes
3    integer: Number of Edge Merges = 0
4
5    for every vertex V in E:
6      for every edge Ev in V:
7        // Ev is the collapsed edge
8        if Ev is E:
9          continue
10
11       // Temporarily replace the vertex that would be removed
12       // with dummy vertex to calculate new hash
13       in edge Ev replace V with dummy vertex
14       integer64: Hash = get hash for Ev
15       if Hash exists in Edge Hashes:
16         Number of Edge Merges += 1
17       else
18         add Hash to Edge Hashes
19
20       if Number of Edge Merges > 2:
21         return true
22
23   // There should be exactly two edge merges
24   if Number of Edge Merges != 2:
25     return true
```

Program 4.19 *Algorithm to check if the edge collapse would result to a non-manifold surface.*

4.4.7 Updating mesh topology at edge collapse

Now we create a new vertex v_n at the edge collapse target and mark all the triangles associated with the edge as removed. For each edge endpoint (v_1, v_2), we go through the edges the vertex is connected to. We replace the vertex with the new vertex v_n. The same procedure is also performed for triangles. Program 4.20 demonstrates the edge and triangle update. In lines 5-14, the edges in decimated vertex are updated to the v_n. In lines 15-25, the triangles connected to the decimated vertices are updated to use v_n. Also the triangle normal vector is computed using the new vertex.

```
1  function update edges and triangles of edge E with new vertex Vn:
2    // Loop through decimated vertices
3    for vertex V in edge E:
4      // Loop through edges connected to decimated vertex
5      for edge Ev in vertex V:
6        // Edge which is removed does not need to be updated
7        if Ev is removed:
8          continue
9
10       // Update edge with Vn
11       in edge Ev replace V with Vn
12       // Associate this edge with new vertex
13       add edge index of Ev for vertex Vn
14
15     for triangle T in vertex V:
16       // No need to update removed triangle
17       if T is removed:
18         continue
19
20       // Update triangle with new vertex Vn and
21       // recalculate normal
22       in triangle T replace V with Vn
23       update normal for T
24       // Associate this triangle with new vertex
25       add triangle index of T for vertex Vn
```

Program 4.20 *Algorithm to update edges and triangles with a new vertex.*

After updating the edges, the new vertex can have edges with same endpoints but with different references to triangles. To fix it, we merge the data of edges with the same endpoints. In Program 4.21, we use a hash map to map unique edges. We iterate through the edges in the new vertex and try to find identical edges using the hash. As previously explained, in Program 4.10, the hash is built from the edge endpoint indices. As shown in Program 4.21 lines 8-14, if a duplicate edge E_d (an edge with same endpoints) is found, the edge E is removed from the edge heap, all its triangle references are added to another edge E_d, and all the references to removed triangle are removed. Otherwise, if an edge is not found, E is added to the hash map. Finally, in lines 18-20, references to decimated edges are removed from vertices referencing to it.

```
1  function merge similar edges in vertex V:
2    hash map of edge: unique edges
3    list of edge: edges decimated
4
5    for edge E in triangle V:
6      integer64: E hash = get hash for E
7      find edge Ed from unique edges with E hash
8      if Ed found:
9        remove E from edge heap
10       add all unique triangle references from edge E to edge Ed
11       remove all removed triangle references from edge Ed
12       add edge E to edges decimated
13     else:
14       add edge E to unique edges with E hash
15
16   // Remove references to removed duplicate edges from vertices
17   // which reference to it
18   for edge E in edges decimated:
19     for every vertex V in E:
20       remove edge E from vertex V
```

Program 4.21 *Algorithm to update edges and triangles with new vertex.*

Now that the data structure has been updated, we update the Q value for the new vertex. The new Q value is directly a sum of the vertices Q matrices that were part of the removed edge. We also update collapse costs and targets of the edges to which the new vertex is connected to. The algorithm is defined in Program 4.22.

```
1  function set Q and compute edge collapse costs for vertex V:
2    Q for V = Q at removed vertex V1 + Q at removed vertex V2
3    for every edge E in vertex V:
4      update collapse cost and target for edge E
5      update edge heap for edge E
```

Program 4.22 *Algorithm to update the matrix Q and compute the edge collapse cost and target for a vertex.*

4.5 Rebuilding the geometric object

After the edge decimation process, we transform the mesh from the decimation data structures back to geometric object G as an indexed triangle mesh. We iterate through the triangles T while adding the vertices and indices to the geometric object structure based on the triangle material. The procedure is shown in detail in Program 4.23. As the vertices are shared between triangles in the decimation data

structure, we split the vertices by triangle surface normal. Currently, we use 30 degrees as a threshold. This is shown in lines 13-23, where indices of triangle mesh vertices which are already added to the triangle mesh with the vertex V index are attempted to be found. We iterate the found vertices. If an angle between the found vertex and the current triangle is within the threshold, then the vertex is reused, and the normal is averaged. Lines 25-31, show how to a case when the vertex is not found either by not being already added to the list or by not reaching the normal angle threshold is handled.

```
1  function rebuild triangle mesh
2    geometric object: G
3
4    for triangle T in triangles:
5      if T is removed:
6        continue
7
8      triangle mesh: Tm = G get mesh for T material
9
10     for vertex V in triangle T:
11       boolean: vertex found = false
12
13       index list: Vl = find vertex indices added to Tm with V index
14       for index I in Vl:
15         // Re-use vertex if normal close to threshold.
16         vector3: vertex normal = Tm normal at I
17         float: angle = angle between vertex normal and T normal
18         if angle < 30:
19           add I to Tm indices list
20           // update the vertex normal by averaging
21           vector3: new normal = (vertex normal + T normal) / 2
22           vertex normal = normalize new normal
23           vertex found = true
24
25       if vertex found == false:
26         add V position to Tm position list
27         add T normal to Tm normal list
28         add T UV coordinates to Tm UV coordinate list
29
30         integer: index = size of Tm position list
31         add index to Tm indices list
32
33   return G
```

Program 4.23 *Algorithm to rebuild the triangle mesh.*

5. RESULTS

In this chapter, we test the implementation described in the previous chapter using various techniques. First, we test the geometric quality of our simplification against Garland and Heckbert reference implementation of QEM based edge decimation in open-source QSlim 2.1 software [65]. Then, the edge collapse cost threshold is tested in practice using Vertex CAD models. Finally, we compare the rendering performance of original models and the simplified approximation in desktop and mobile environments.

5.1 Simplification quality

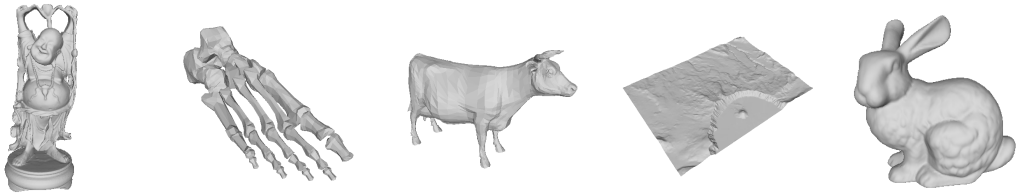
We use geometric quality in order to test the simplification quality and validate our implementation against a reference implementation. To measure geometric quality between the original and approximation, we use Hausdorff distance as a measure. We use open-source MeshLab 2016.12 software to calculate the Hausdorff distance [66].

5.1.1 Difference to reference implementation

We compare our implementation to QSlim software output. The QSlim software reads only in *simple model format* (smf) which seems to be an unknown format, and as a result, we found very few models using the format. The smf format specification is part of the QSlim software [65]. The smf format is a subset of the Wavefront obj format, which Vertex CAD software supports as an import file format [67]. As the QSlim software outputs the simplified geometry in smf format, we modified the export pipeline to output the simplified mesh from Vertex CAD software in the same format and accuracy to make the comparison as fair as possible. The QSlim software is run with default settings.

Fortunately, for this test, we use models provided by Garland on his website [68]. These same models are the same used in Garland and Heckbert papers [38, 39]. From that model set, we use models buddha, skeletal foot, cow, crater lake and

bunny, which are illustrated in Figure 5.1. We have decided to use four different levels of decimation amount in order to test the quality in a variety of approximation levels. The selected levels are to reduce 50, 70, 90 and 95 percent of the original face count. For this test, we also disabled the target cost threshold in our algorithm to match the simplification levels of QSlim output. For these models, Hausdorff distance measurement is configured to sample 10 million points from the original mesh faces and find the nearest point to the simplified approximation mesh. The detailed results of the measurements are in appendix A.



(a) Buddha (b) Skeletal foot (c) Cow (d) Crater lake (e) Bunny

Figure 5.1 *The collection of models that are evaluated.*

In Table 5.1 we have results comparing the root mean square (RMS) Hausdorff distance between the QSlim and Vertex CAD simplification outputs M_{QSlim} and M_{Vertex}. The distance is relative to the diagonal of the model's bounding box. The difference is calculated as $D_{rmsdiff} = D_{rmsM_{QSlim}} - D_{rmsM_{Vertex}}$. Negative difference means that the QSlim output has lower RMS Hausdorff distance and vice versa. In the table, the difference is scaled by 100 to improve the readability of the results as we are interested only the difference between the two outputs. That also means that the reported error difference is a percentage of the model's bounding box diagonal.

Table 5.1 *Scaled relative difference of **RMS** Hausdorff distance between the QSlim and Vertex output using different levels of face count reduction. A positive value is in favor of Vertex output. The value can be interpreted as a percentage of the model's bounding box diagonal.*

Model	**50%**	**70%**	**90%**	**95%**
Skeletal foot (4.2k triangles)	-0,0125	-0,0226	0,271	0,3643
Buddha (1085k triangles)	0,0001	0,0002	0,0014	0,0034
Bunny (69k triangles)	0,0015	0,0035	0,0151	0,0341
Cow (5.8k triangles)	-0,0083	-0,0137	-0,0207	-0,0407
Crater (119k triangles)	-0,0002	-0,0003	-0,0006	-0,0013

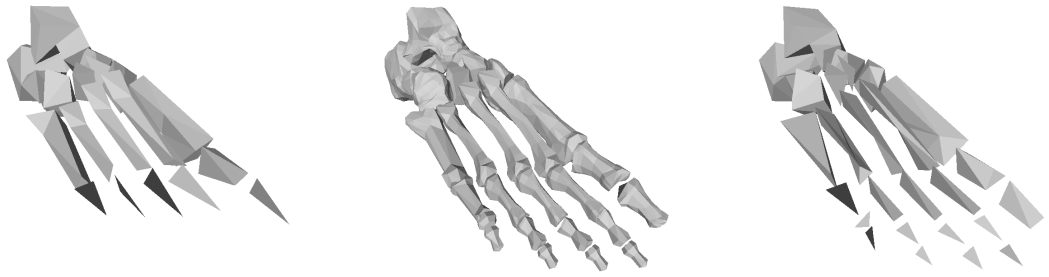
In Table 5.2 we show the maximum difference of Hausdorff distance $D_{maxdiff} = D_{maxM_{QSlim}} - D_{maxM_{Vertex}}$. The result is also scaled in the same way as for the previous table and thus it is a percentage of the model's bounding box diagonal.

Table 5.2 *Scaled relative difference of* ***maximum*** *Hausdorff distance between the QSlim and Vertex output using different levels of face count reduction. A positive value is in favor of Vertex output. The value can be interpreted as a percentage of the model's bounding box diagonal.*

Model	**50%**	**70%**	**90%**	**95%**
Skeletal foot (4.2k triangles)	0,1305	0,0002	5,2827	6,4497
Buddha (1085k triangles)	0,1826	0,2332	0,5811	0,5584
Bunny (69k triangles)	0,0151	0,022	0,0182	0,1103
Cow (5.8k triangles)	-0,0464	-0,0486	0,6143	0,4542
Crater (119k triangles)	0,0002	-0,0008	-0,0294	-0,1421

From the results, we can see that overall we are close to the results of QSlim output. For the skeleton foot model, we have much improved maximum Hausdorff distance and for the others marginally better. The difference is most likely to be attributed to face area weighting which QSlim uses by default. Investigating the implementation detail, they multiply the vertices quadric error matrices by the sum of the triangle areas connected to the vertex.

From Figure 5.2, the difference of the skeletal foot mesh is seen as the output from our implementation has preserved more detail on minor parts whereas QSlim has deleted some details. The difference is explained by the non-manifold constraints that we have enforced. The constraint rejects edge decimation operations when it would result in non-manifold geometry. The disadvantage is that the edge has to be removed some other place that has greater decimation cost in order to reach the same decimation level.



(a) 95% of faces decimated using QSlim.

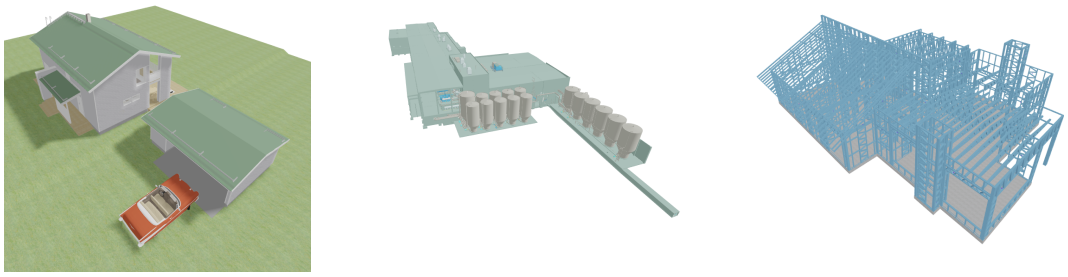
(b) Original mesh.

(c) 95% of faces decimated using our implementation.

Figure 5.2 *Comparison between the simplified skeletal foot mesh outputs of QSlim, our implementation, and the original mesh.*

5.1.2 Evaluating edge collapse cost threshold

We selected three different types of Vertex CAD models to evaluate the edge collapse cost threshold T_c defined in Chapter 4. The first model is a plant design model, which contains lots of planar surfaces and some amount of round surfaces. The second model is a building design model that contains lots of round surfaces and some amount of planar surfaces. The last model is a building framing model that mostly contains planar surfaces. Figure 5.3 illustrates the models that are used for the evaluation. Vertex Systems Oy provided the Vertex CAD models evaluated in this test.



(a) Building design. (b) Plant design. (c) Framing design.

Figure 5.3 *The collection of Vertex CAD models that are evaluated.*

For this test, we set the edge collapse cost scale to 0,0001 and the face reduction threshold is 90%. With our definition, theoretically, the maximum error of single edge collapse operation has to be under 0,1% of the model's bounding box diagonal. We selected those values based on the output which were on par with the quality and simplification amount we wanted. To measure Hausdorff distance, we utilize the MeshLab software and compute 40 million samples from faces of the original mesh to the simplified mesh. The models are exported from Vertex CAD software to glTF file format.

Table 5.3 presents the results of the measurements. Appendix B contains detailed images comparing the simplification to the original model. We see the effect of the threshold parameter, as the framing design model featuring mostly planar surfaces does not benefit from the simplification at all. That is illustrated better in Figure B.1, which shows the planar surfaces of the mesh. Furthermore, the building model, which contains lots of round surfaces benefits considerably from the simplification. For example, this can be seen from Figure B.2 where the door handle has been simplified a considerable amount. Besides, as shown in Figure B.3, decorative objects benefit also to a great extent. Finally, the plant design model is on the middle and

ends up benefiting comparatively little. A further investigation revealed that the plant model contains lots of pipe parts which are represented using non-manifold meshes. On straight piping sections, the mesh consist of single-sided rounded cylinder in which every vertex lies on a boundary edge. Based on our constraints the mesh did not get simplified at all. That is illustrated in Figure B.5. Similarly to building simplification result, objects which benefit from the simplification are constructed from round elements, as Figure B.4 illustrates.

Table 5.3 *Edge decimation results of Vertex CAD models using the edge collapse cost threshold.*

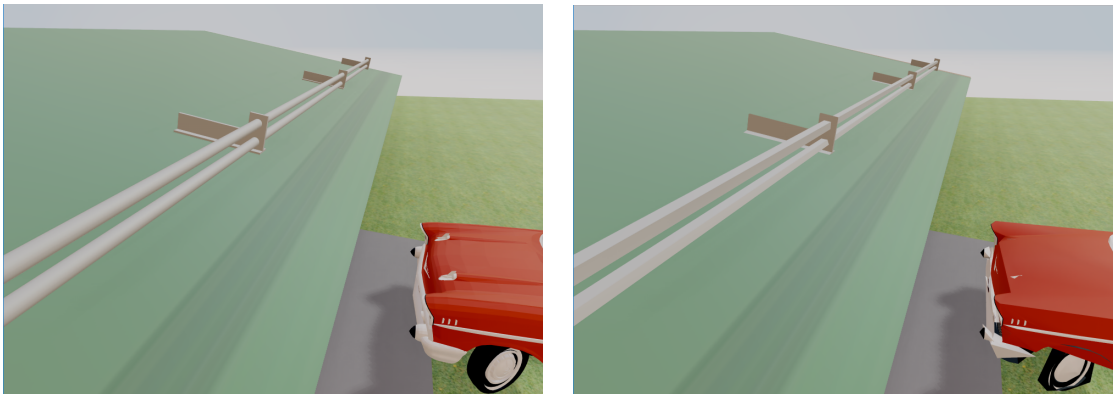
Model	Face count original	Face count simplified	Realized face count reduction (%)	RMS Hausdorff distance	Max Hausdorff distance
Plant	12 625 658	8 559 494	32,2	0,002429	0,115265
Building	15 297 934	1 852 203	87,9	0,003351	0,239714
Framing	695 784	685 272	1,5	0,000006	0,001011

The binary buffer size of the resulting glTF file contains geometry and texture data. In Table 5.4 we list the resulting buffer size of with and without simplification. The results show that the buffer size of the framing design model has grown after simplification. That is the result of the simplification algorithm splitting vertices by normal when the simplified mesh is rebuilt (Program 4.23). As the simplification factor is low, the growth in vertices causes the simplified mesh to grow in buffer size.

Table 5.4 *Effect to buffer size of Vertex CAD models after simplification using the edge collapse cost threshold.*

Model	Buffer size original (bytes)	Buffer size simplified (bytes)	Realized buffer size reduction (%)
Plant	569 076 844	399 757 500	29,8
Building	481 994 244	67 702 780	86,0
Framing	42 208 228	44 297 028	-4,9

Figure 5.4 shows a comparison between the original and the simplified building design models. From the image, an apparent reduction in geometry is seen in the car model and on the railing at the roof. On the car model, the hood has been flattened, and tires are noticeably more planar. On the roof, the railing has been turned from perfectly round to a much more planar shaped object.



(a) Original model. (b) Simplified model.

Figure 5.4 *A comparison between the original building design model and the simplified model outputs from Vertex CAD.*

5.2 Rendering performance

We use frame time as a metric to evaluate the difference of rendering performance between the original and the simplified model. The rendering performance test is performed on a desktop computer and on a mobile device. In both tests, we collect and analyze 5000 samples of frame times. We use the same three models designed and exported from the Vertex CAD program as in the previous test. An application executable for both desktop and mobile environments are created using the Unity 2018.2.17f1 game engine [69]. The application has a scene containing the mesh that is measured. In our Unity scene setup, all environment effects such as environment map, lights, shadows are disabled to reduce additional processing time during frame rendering. Such effects are unrelated to the actual mesh rendering of which performance we are interested in. In order to focus on performance differences on vertex processing and reduce the effects of pixel processing performance, we set the materials to unlit.

5.2.1 Desktop environment

The desktop computer used to execute the application is using Windows 10 Enterprise (1803) operating system and has an Intel Core i7-7700K processor with 32 gigabytes of DDR4 memory. The graphics processing unit is NVIDIA Quadro K1200, and the graphics driver version is 384.0. The computer configuration was selected as it was readily available. During frame time measurement, the application is run in a full-screen mode with a resolution of 640x480. The low resolution is selected in order to minimize the effect of pixel processing to the overall frame time.

We use *The Open Capture and Analytics Tool* (OCAT) for measuring the frame time of the application [70]. During the testing, no user input was performed, and the number of open applications was minimized to reduce the external performance impact to the testing.

Table 5.5 includes the results from the test. We can see that the performance almost scales by the number of faces decimated. For the building design model, the performance improved dramatically, allowing the model, for example, to be visualized in a virtual reality environment with good performance.

Table 5.5 *Average frame times recorded on desktop computer using original and simplified Vertex CAD models.*

Model	Average frame time original (ms)	Average frame time simplified (ms)	performance scale
Plant	12,3	8,6	1.43
Building	24,4	3,2	7.63
Framing	1,2	1,2	0.00

5.2.2 Mobile environment

The same application is executed in LG G2 mobile phone, which is using the Android 5.0.2 operating system. The device was selected as it was easily available for testing and provides a reference to low-end Android phone. As Android surface presenter has vertical synchronization always enabled, the frame time is clamped to it [8]. That means that a detailed analysis of frame time is impossible without using a GPU profiler. We collect the frame times using an internal program that collects the current frame time to an array. Finally, it saves the collected frame times to a file after a specific amount of frames has been recorded. Table 5.6 shows the average frame times for the mobile environment.

Table 5.6 *Average frame times recorded on mobile computer using original and simplified Vertex CAD models.*

Model	Average frame time original (ms)	Average frame time simplified (ms)
Plant	N/A	N/A
Building	N/A	32.2
Framing	18.4	18.5

The plant design model was unable to load both original and simplified versions due to memory constraints. For the building design model, the original model was

unable to load. Fortunately, the substantial reduction in geometry the simplified model allowed the mesh to load and run at around 30 frames per second (13.3 milliseconds frame time). For the framing design model, no performance differences were measured as expected.

6. CONCLUSIONS

This thesis reviewed methods of simplifying 3D meshes using offline decimation algorithms. New constraints for rendering performance and mesh complexity have been added, as the visualization of 3D CAD models on new platforms such as mobile and virtual reality are becoming more significant. A general simplification is a hard problem as the input mesh might vary a lot in complexity, size, and form. A balance is needed between the visual quality and rendering performance which we are attempting to achieve.

We defined and built a custom data structure in order to modify and collapse edges easily. We then implemented an iterative edge decimation algorithm based on quadric error metric proposed by Garland and Heckbert [38]. A face count target threshold works poorly as a condition to terminate the decimation iteration when the complexity of the input meshes varies a lot. For example, a relatively simple model consisting of planar faces cannot be decimated as much as a model with round surfaces without having a significant impact on quality. To aid such situations, we proposed an edge collapse cost threshold T_c which is calculated by multiplying a user-defined scale variable and the diagonal of the decimated model's bounding box. The threshold T_c represents a theoretical maximum for possible error that is allowed during the decimation process. Finally, after the decimation process is terminated, we rebuilt the mesh from the custom data structure to an indexed triangle mesh.

We evaluated the quality and performance of the simplification algorithm. First, we compared our implementation against the QSlim software output using root mean square and the maximum value of Hausdorff distance. Overall the differences to QSlim output are small. The differences of RMS values are well under 1 percent relative to the diagonal of mesh's bounding box. An apparent exception was the skeleton foot mesh where our implementation managed to retain more high-frequency details while having a high face reduction level. That was shown by the significant difference in maximum Hausdorff distance and, furthermore, is seen in Figure 5.2. Next, we evaluated the performance of the edge collapse cost threshold T_c. By using three different Vertex CAD models, we saw an expected result of significant face count reduction in situation where the model contains lots of round surfaces with

high-frequency detail. Vice versa, the framing design model which contained lots of low-frequency detail did not benefit from the simplification as in that case the error would rise in a fast rate. Finally, we measured performance improvements which are gained from the simplification. From the results, we could see that the performance scaled almost the same rate as the decimation amount. We managed to get the complex building design model to run on low-end mobile by simplifying it a significant factor. All in all, the threshold T_c works as expected, it terminates the decimation before the error raises too high and it scales with the model size. As a downside, it still needs a user-defined scale value, which is configured for the user's preference. Based on the results, it does not work well for models with mostly planar faces.

Overall, the thesis process went well. The project started by researching mesh simplification methods. Then, implementing the iterative edge decimation algorithm based on quadric error metric proposed by Garland and Heckbert to the Vertex CAD software [38]. As the algorithm performed poorly with fixed settings, we added the edge collapse cost threshold, which was simple to implement and understand. In hindsight, an external proof of concept program would have made the testing process faster, in addition to improving the time of developing, modifying and validating the simplification algorithm. Conversely, the described simplification is now in use in Vertex CAD software.

6.1 Future work

For future improvements, the threshold T_c should be tested further using various new scale values. Instead of using the diagonal of the model's bounding box, other ways of computing the threshold could be investigated as well. In order to improve the visual quality, the decimation algorithm should take account textures and also handle the transformation of the texture coordinates. For example, texture errors can be seen in Figure B.3, where the green texture is skewed after simplification. Garland and Heckbert extent the quadric error metric to use the texture coordinates and other possible vertex attributes in their work [39]. Hoppe also researches the same extension [42].

A displacement or normal map could be generated from the detailed model and applied to the simplified mesh. The normal map would then give the mesh an approximation of the removed detail. Normal maps were successfully used by Cohen et al. to preserve appearance [57]. A possible issue for normal map generation is the memory usage of the generated maps when the input CAD scene is large. In order to shorten the simplification time, the algorithm could be task parallelized for

multiple geometric objects $M = \{m_0, m_1, ...m_i\}$. As the algorithm does not share any state between the objects, each simplification is an independent task. Currently, the algorithm does not utilize geometric object instancing, where objects containing identical geometry with different transformation are simplified only once, and the simplification result is reused for the rest of the instanced objects. That would significantly speed up scenes where the same object is shared multiple times.

For run-time performance and visual quality improvements, a level-of-detail hierarchy could be built. That means that a predetermined number of approximations with different face reduction amounts are generated per model basis. A large CAD scene could benefit from the LOD hierarchy, as models close to the camera's viewpoint would be shown in higher resolution, whereas models at further would be shown in ascending levels of lower resolution depending on the distance to the viewpoint.

BIBLIOGRAPHY

[1] Vertex Systems Oy, Flickr, Public image gallery, Source (fetched 08.09.2018): https://www.flickr.com/photos/130657952@N03/.

[2] P. Milgram, A. Kishino, Taxonomy of Mixed Reality Visual Display, IEICE Transactions on Information Systems, Vol E77-D, 1994.

[3] J. Arvo, A. Euranto, L. Järvenpää, T. Lehtonen, T. Knuutila, 3D mesh simplification - A survey of algorithms and CAD model simplification tests, University of Turku Technical Reports, No. 3, August 2015.

[4] I. Stroud, Boundary Representation Modelling Techniques, 1st ed. Springer Verlag London Limited, 2006, pp. 1-8.

[5] K. Rutanen, Half-edge structure, May 2007, Source (fetched 28.06.2018): http://kaba.hilvi.org/homepage/blog/halfedge/halfedge.htm.

[6] M. Garland, Quadric-Based Polygonal Surface Simplification, Ph.D. dissertation, Computer Science Department, Carnegie Mellon University, 1999, Source (fetched 12.11.2018): https://mgarland.org/research/thesis.html.

[7] M. Abrash, What VR could, should, and almost certainly will be within two years, Steam Dev Days, 2014, Seattle.

[8] SurfaceFlinger and Hardware Composer, Android Open Source Project, June 2018, Source (fetched 26.06.2018): https://source.android.com/devices/graphics/arch-sf-hwc.

[9] CADisplayLink - Core Animation, Apple, 2018, Source (fetched 26.06.2018): https://developer.apple.com/documentation/quartzcore/cadisplaylink.

[10] M. Sandy, Announcing Microsoft DirectX Raytracing!, Microsoft, March 2018, Source (fetched 12.11.2018): https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/.

[11] N. Subtil, Introduction to Real-Time Ray Tracing with Vulkan, Nvidia, October 2018, Source (fetched 12.11.2018): https://devblogs.nvidia.com/vulkan-raytracing/.

[12] NVIDIA Turing GPU Architecture, NVIDIA Corporation, 2018, Whitepaper, Source (fetched: 14.12.2018): https://www.nvidia.com/

content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[13] Y. Deng, Y. Ni, Z. Li, S. Mu, W. Zhang, Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques, ACM Computing Surveys (CSUR), Vol. 50, No 4, 2017, pp. 1-41.

[14] Single Pass Stereo rendering, Unity Technologies, 2018, Documentation, Source (fetched 14.12.2018): https://docs.unity3d.com/Manual/SinglePassStereoRendering.html.

[15] T. Poulet, Optimizing Virtual Reality: Understanding foveated rendering, Arm Ltd, 2017, Source (fetched 14.12.2018): https://community.arm.com/graphics/b/blog/posts/optimizing-virtual-reality-understanding-foveated-rendering.

[16] M. Mikkelsen, Simulation of Wrinkled Surfaces Revisited, Thesis, 2008, Source (fetched 16.11.2018): http://image.diku.dk/projects/media/morten.mikkelsen.08.pdf.

[17] M. Bunnell, Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping, GPU Gems 2, Chapt. 7, Source (fetched 16.11.2018): https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter07.html.

[18] 3D ACIS Modeling, Dassault Systèmes, 2018, Source (fetched 12.11.2018): https://www.spatial.com/products/3d-acis-modeling.

[19] P. Cignoni, C. Montani, R. Scopigno, A comparison of mesh simplification algorithms, Computer & Graphics, Vol. 22, No. 1, 1998, pp. 37-54.

[20] R. Klein, G. Liebich, W. Straßer, Mesh Reduction with Error Control, Proceedings of the 7th Conference on Visualization, 1996, pp. 311-318.

[21] J. Rossignac, P. Borrel, Multi-resolution 3D approximations for rendering complex scenes, Geometric Modeling in Computer Graphics, June-July 1993, pp. 455-465.

[22] P. Lindstrom, Out-of-Core Simplification of Large Polygonal Models, Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, 2000, pp. 259-262.

[23] C. DeCoro, N. Tatarchuk, Real-Time Mesh Simplification using the GPU, Proceedings of the 2007 Symposium on Interactive 3d Graphics and Games, 2007, pp. 161-166.

[24] E. Shaffer, M. Garland, Efficient adaptive simplification of massive meshes, Proceedings of the Conference on Visualization, 2001, pp. 127-134.

[25] S. Schaefer, J. Warren, Adaptive Vertex Clustering Using Octrees, SIAM Geometric Design and Computing, 2003, pp. 491–500.

[26] T. Boubekeur, M. Alexa, Mesh Simplification by Stochastic Sampling and Topological Clustering, Computers & Graphics, Vol. 33, No. 3, 2009, pp. 241-249.

[27] D. Brodsky, B. Watson, Model Simplification Through Refinement, Graphics Interface, 2000, pp. 221-228.

[28] D. Brodsky, J. Pedersen, A Parallel Framework for Simplification of Massive Meshes', IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003, pp. 17-24.

[29] M. Limper, Y. Jung, J. Behr, M. Alexa, The POP Buffer: Rapid Progressive Clustering By Geometry Quantization, Computer Graphics Forum, Vol. 32, No. 7, 2013, pp. 197-206.

[30] W. J. Schroeder, J. A. Zarge, W. E. Lorensen, Decimation of Triangle Meshes, Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques, 1992, pp. 65-70.

[31] M. Soucy, D. Laurendeau, Multiresolution Surface Modeling Based on Hierarchical Triangulation, Computer Vision and Image Understanding, Vol. 63, No. 1, 1996, pp. 1-14.

[32] C. Chuon, S. Guha, Volume Cost Based Mesh Simplification, 2009 Sixth International Conference on Computer Graphics, Imaging and Visualization, 2009, pp. 164-169.

[33] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle, Mesh optimization, Proceedings of the ACM SIGGRAPH, 1993, pp. 19-26.

[34] H. Hoppe, Progressive Meshes, Proceedings of the ACM SIGGRAPH, 1996, pp. 99-108.

[35] H. Hoppe, View-Dependent Refinement of Progressive Meshes, Proceedings of the ACM SIGGRAPH, 1997, pp. 189-198.

[36] M. Franc, V. Skala, Fast Algorithm for Triangular Mesh Simplification Based on Vertex Decimation, Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 2330, No. 2, 2002, pp. 42-51.

[37] R. Ronfard, J. Rossignac, Full-range approximation of triangulated polyhedra. Computer Graphics Forum, Vol. 15, No. 3, 1996, pp. 67-76.

[38] M. Garland, P. Heckbert, Surface Simplification using Quadric Error Metrics, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, 1997, pp. 209-216.

[39] M. Garland, P. Heckbert, Simplifying Surfaces with Color and Texture using Quadric Error Metrics, Proceedings of the Conference on Visualization, 1999, pp. 263-269.

[40] K. Bahirat, C. Lai, R. Mcmahan, B. Prabhakaran. Designing and Evaluating a Mesh Simplification Algorithm for Virtual Reality, ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), Vol. 14, No. 3, 2018, pp. 1-26.

[41] P. Lindstrom, G. Turk, Fast and Memory Efficient Polygonal Simplification, Proceedings of the Conference on Visualization, 1998, pp. 279-286.

[42] H. Hoppe, New Quadric Metric for Simplifying Meshes with Appearance Attributes, Proceedings of the IEEE Visualization Conference, 1999, pp. 59-66.

[43] D. Salinas, F. Lafarge, P. Alliez, Structure-Aware Mesh Decimation, Computer Graphics Forum, Vol. 34, No. 6, 2015, pp. 211-227.

[44] A. Papageorgiou, N. Platis, Triangular Mesh Simplification on the GPU, The Visual Computer, Vol. 31, No. 2, 2010, pp. 235–244.

[45] L. Hu, P. Sander, H. Hoppe, Parallel view-dependent refinement of progressive meshes, Symposium on Interactive 3D Graphics and Games, 2009, pp. 169-176.

[46] L. Hu, P. Sander, H. Hoppe, Parallel view-dependent level-of-detail control, IEEE Transactions on Visualization and Computer Graphics, Vol. 16, No. 5, 2010, pp. 718-728.

[47] T. Odaker, D. Kranzlmueller, J. Volkert, View-dependent Triangle Mesh Simplification using GPU-accelerated Vertex Removal, The 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, 2016.

[48] H. Lee, M. Kyung, Parallel mesh simplification using embedded tree collapsing, The Visual Computer, Vol. 32, No. 6, 2016, pp. 967-976.

[49] B. Hamann, A data reduction scheme for triangulated surfaces, Computer Aided Geometric Design, Vol. 11, No. 2, 1994, pp. 197-214.

[50] T. Gieng, B. Hamann, K. Joy, G. Schussman, I. Trotts, Smooth Hierarchical Surface Triangulations, Proceedings of the 8th conference on visualization, 1997, pp. 379-386.

[51] T. Gieng, B. Hamann, K. Joy, G. Schussman, I. Trotts, Constructing Hierarchies for Triangle Meshes, IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 2, 1998, pp. 145-161.

[52] V. Isler, R. Lau, M. Green, Real-time Multi-resolution Modeling for Complex Virtual Environments, Symposium on Virtual Reality Software and Technology, 1996, pp. 11-20.

[53] P. Lindstrom, G. Turk, Image-Driven Simplification, ACM Transactions on Graphics, Vol. 19, No. 3, Jul 2000, pp. 204–241.

[54] E. Zhang, G. Turk, Visibility-Guided Simplification, Proceedings of the Conference on Visualization, 2002, pp. 267-274.

[55] C. Lee, A. Varshney, D. Jacobs, Mesh Saliency, ACM Transactions on Graphics, 2005, pp. 659-666.

[56] L. Qu, G. Meyer, Perceptually Guided Polygon Reduction, IEEE Transactions on Visualization and Computer Graphics, Vol. 14, No. 5, 2008, pp. 1015-1029.

[57] J. Cohen, M. Olano, D. Manocha, Appearance-preserving simplification, ACM Transactions on Graphics, 1998, pp. 115-122.

[58] R. Álvarez, J. Noguera, L. Tortosa, A. Zamona, A Mesh Optimization Algorithm Based on Neural Networks, Information Sciences, Vol. 177, No. 23, 2007, pp. 5347-5364.

[59] C. González , J. Gumbau, M. Chover, F. Ramos, R. Quirós, User-assisted simplification method for triangle meshes preserving boundaries, Computer-Aided Design, Vol. 41, No. 12, 2009, pp. 1095-1106.

[60] S. Gao, W. Zhao, H. Lin, F. Yang, X. Chen, Feature suppression based CAD mesh model simplification, Vol. 42, No. 12, 2010, pp. 1178-1188.

[61] S. Kanai, D. Iyoda, Y. Endo, H. Sakamoto, N. Kanatani, Appearance preserving simplification of 3D CAD model with large-scale assembly structures, International Journal on Interactive Design and Manufacturing, 2012, pp. 139-154.

[62] glTF Overview, The Khronos Group Inc, 2018, Website, Source (fetched 16.11.2018): `https://www.khronos.org/gltf/`.

[63] glTF-SDK, Microsoft, Open-source library, MIT licence, Source (fetched 16.11.2018): https://github.com/Microsoft/glTF-SDK.

[64] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, M. Gross, Optimized Spatial Hashing for Collision Detection of Deformable Objects, Proceedings of Vision, Modeling, Visualization, 2003, pp. 47-54.

[65] M. Garland, QSlim 2.1, 2004, Open-source software, GPLv2 license, Source (fetched 19.11.2018): https://mgarland.org/software/qslim.html.

[66] Meshlab: an open-source mesh processing tool, P. Cignoni, M. Callieri, M. Dellepiane, F. Ganovelli, G. Ranzuglia, Proceedings of the Eurographics Italian Chapter Conference, 2008, pp. 129-136.

[67] Wavefront .obj format, Specification, Source (fetched 10.12.2018): https://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf.

[68] M. Garland, Sample data, 3D models, Source (fetched 20.11.2018): https://mgarland.org/research/quadrics.html.

[69] Unity, Unity Technologies, Source (fetched 8.12.2018): https://unity3d.com.

[70] The Open Capture and Analytics Tool (OCAT), GPUOpen, Open-source software, MIT license, Source (fetched 8.12.2018): https://github.com/GPUOpen-Tools/OCAT.

APPENDIX A. HAUSDORFF DISTANCE RESULTS

This appendix contains results of measuring Hausdorff distance by sampling 10 million points from the original mesh to the simplified mesh. The number suffix in the model file name describes the percentage of faces removed. If the model file includes character "g" it means that the model has been decimated using QSlim software, otherwise using our implementation.

```
Sampled 10000000 pts on bones.obj searched closest on bones_50.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.004746   mean : 0.000450   RMS : 0.000623

Sampled 10000000 pts on bones.obj searched closest on bones_g_50.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.006051   mean : 0.000323   RMS : 0.000498

Sampled 10000000 pts on bones.obj searched closest on bones_70.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.007736   mean : 0.000980   RMS : 0.001287

Sampled 10000000 pts on bones.obj searched closest on bones_g_70.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.007738   mean : 0.000768   RMS : 0.001061

Sampled 10000000 pts on bones.obj searched closest on bones_90.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.021050   mean : 0.003070   RMS : 0.003953

Sampled 10000000 pts on bones.obj searched closest on bones_g_90.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.073877   mean : 0.003223   RMS : 0.006663

Sampled 10000000 pts on bones.obj searched closest on bones_95.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.027494   mean : 0.005483   RMS : 0.006910

Sampled 10000000 pts on bones.obj searched closest on bones_g_95.obj
Values w.r.t. BBox Diag (12.603421)
     min : 0.000000   max 0.091991   mean : 0.006003   RMS : 0.010553

Sampled 10000000 pts on buddha.obj searched closest on buddha_50.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.000188   mean : 0.000005   RMS : 0.000007
```

```
Sampled 10000000 pts on buddha.obj searched closest on buddha_g_50.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.002014   mean : 0.000004   RMS : 0.000008

Sampled 10000000 pts on buddha.obj searched closest on buddha_70.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.000406   mean : 0.000013   RMS : 0.000017

Sampled 10000000 pts on buddha.obj searched closest on buddha_g_70.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.002738   mean : 0.000012   RMS : 0.000019

Sampled 10000000 pts on buddha.obj searched closest on buddha_90.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.001295   mean : 0.000041   RMS : 0.000053

Sampled 10000000 pts on buddha.obj searched closest on buddha_g_90.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.007106   mean : 0.000047   RMS : 0.000067

Sampled 10000000 pts on buddha.obj searched closest on buddha_95.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.002505   mean : 0.000075   RMS : 0.000098

Sampled 10000000 pts on buddha.obj searched closest on buddha_g_95.obj
Values w.r.t. BBox Diag (0.229031)
     min : 0.000000   max 0.008089   mean : 0.000093   RMS : 0.000132

Sampled 10000000 pts on bunny.obj searched closest on bunny_50.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.000361   mean : 0.000037   RMS : 0.000050

Sampled 10000000 pts on bunny.obj searched closest on bunny_g_50.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.000512   mean : 0.000045   RMS : 0.000065

Sampled 10000000 pts on bunny.obj searched closest on bunny_70.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.000685   mean : 0.000073   RMS : 0.000096

Sampled 10000000 pts on bunny.obj searched closest on bunny_g_70.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.000905   mean : 0.000099   RMS : 0.000131

Sampled 10000000 pts on bunny.obj searched closest on bunny_90.obj
Values w.r.t. BBox Diag (0.250247)
```

```
     min : 0.000000   max 0.002379   mean : 0.000196   RMS : 0.000256

Sampled 10000000 pts on bunny.obj searched closest on bunny_g_90.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.002561   mean : 0.000322   RMS : 0.000407

Sampled 10000000 pts on bunny.obj searched closest on bunny_95.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.003633   mean : 0.000363   RMS : 0.000475

Sampled 10000000 pts on bunny.obj searched closest on bunny_g_95.obj
Values w.r.t. BBox Diag (0.250247)
     min : 0.000000   max 0.004736   mean : 0.000653   RMS : 0.000816

Sampled 10000000 pts on cow.obj searched closest on cow_50.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.005833   mean : 0.000243   RMS : 0.000338

Sampled 10000000 pts on cow.obj searched closest on cow_g_50.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.005369   mean : 0.000153   RMS : 0.000255

Sampled 10000000 pts on cow.obj searched closest on cow_70.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.007455   mean : 0.000535   RMS : 0.000691

Sampled 10000000 pts on cow.obj searched closest on cow_g_70.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.006969   mean : 0.000390   RMS : 0.000554

Sampled 10000000 pts on cow.obj searched closest on cow_90.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.023186   mean : 0.001666   RMS : 0.002147

Sampled 10000000 pts on cow.obj searched closest on cow_g_90.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.029329   mean : 0.001322   RMS : 0.001940

Sampled 10000000 pts on cow.obj searched closest on cow_95.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.041921   mean : 0.003205   RMS : 0.004320

Sampled 10000000 pts on cow.obj searched closest on cow_g_95.obj
Values w.r.t. BBox Diag (1.271114)
     min : 0.000000   max 0.046463   mean : 0.002630   RMS : 0.003913

Sampled 10000000 pts on crater.obj searched closest on crater_50.obj
```

```
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000109   mean : 0.000010   RMS : 0.000014

Sampled 10000000 pts on crater.obj searched closest on crater_g_50.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000111   mean : 0.000008   RMS : 0.000012

Sampled 10000000 pts on crater.obj searched closest on crater_70.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000235   mean : 0.000020   RMS : 0.000026

Sampled 10000000 pts on crater.obj searched closest on crater_g_70.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000227   mean : 0.000017   RMS : 0.000023

Sampled 10000000 pts on crater.obj searched closest on crater_90.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000962   mean : 0.000050   RMS : 0.000068

Sampled 10000000 pts on crater.obj searched closest on crater_g_90.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.000668   mean : 0.000045   RMS : 0.000062

Sampled 10000000 pts on crater.obj searched closest on crater_95.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.002521   mean : 0.000086   RMS : 0.000119

Sampled 10000000 pts on crater.obj searched closest on crater_g_95.obj
Values w.r.t. BBox Diag (568.698364)
     min : 0.000000   max 0.001100   mean : 0.000075   RMS : 0.000106
```

APPENDIX B. FIGURES OF ORIGINAL AND SIMPLIFIED VERTEX CAD MODELS

This appendix contains pictures of original (up) and simplified (down) Vertex CAD model.

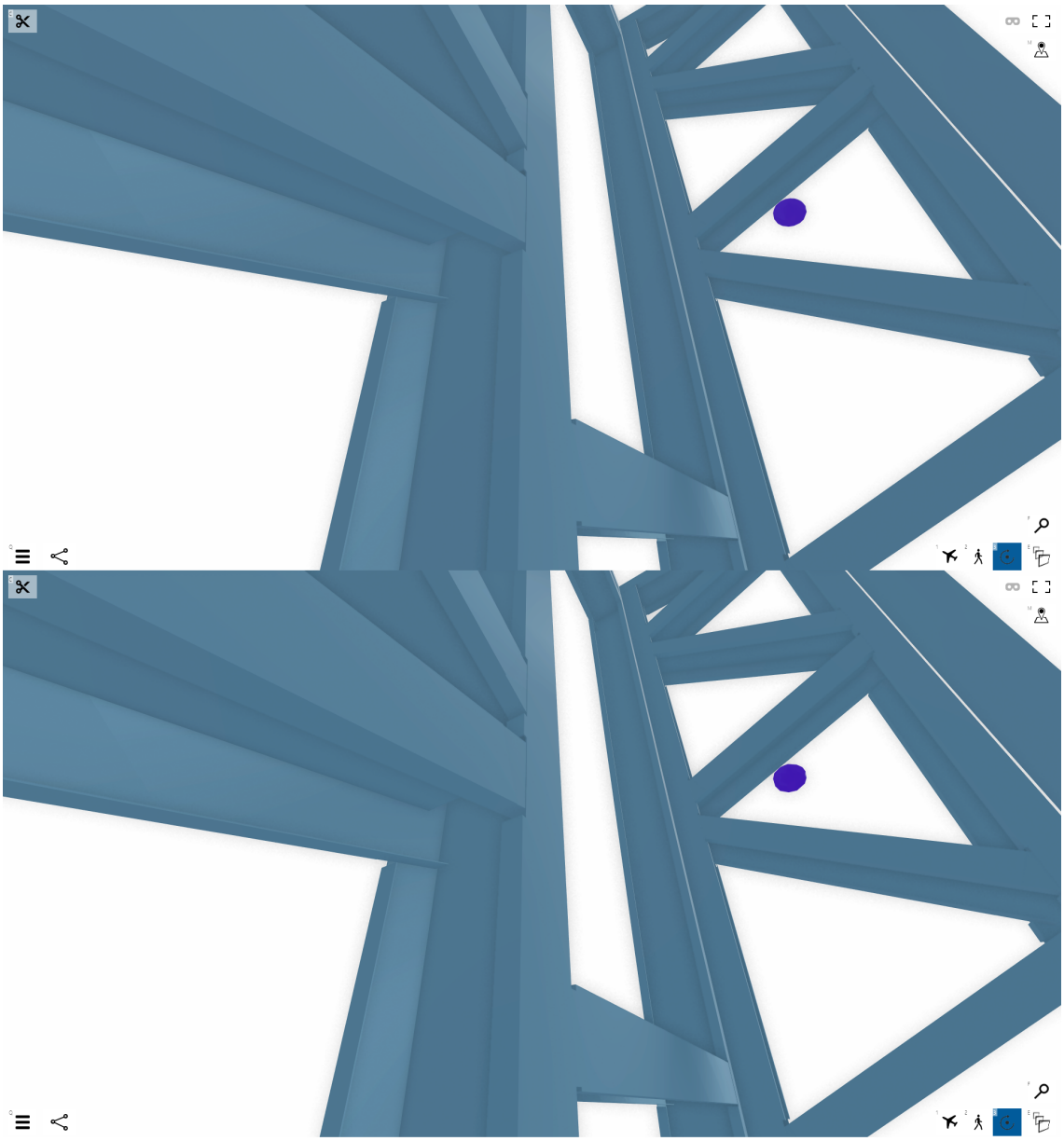


Figure B.1 *Original and simplified Vertex CAD framing model. The model consist of planar surfaces and thus does not benefit from the simplification.*

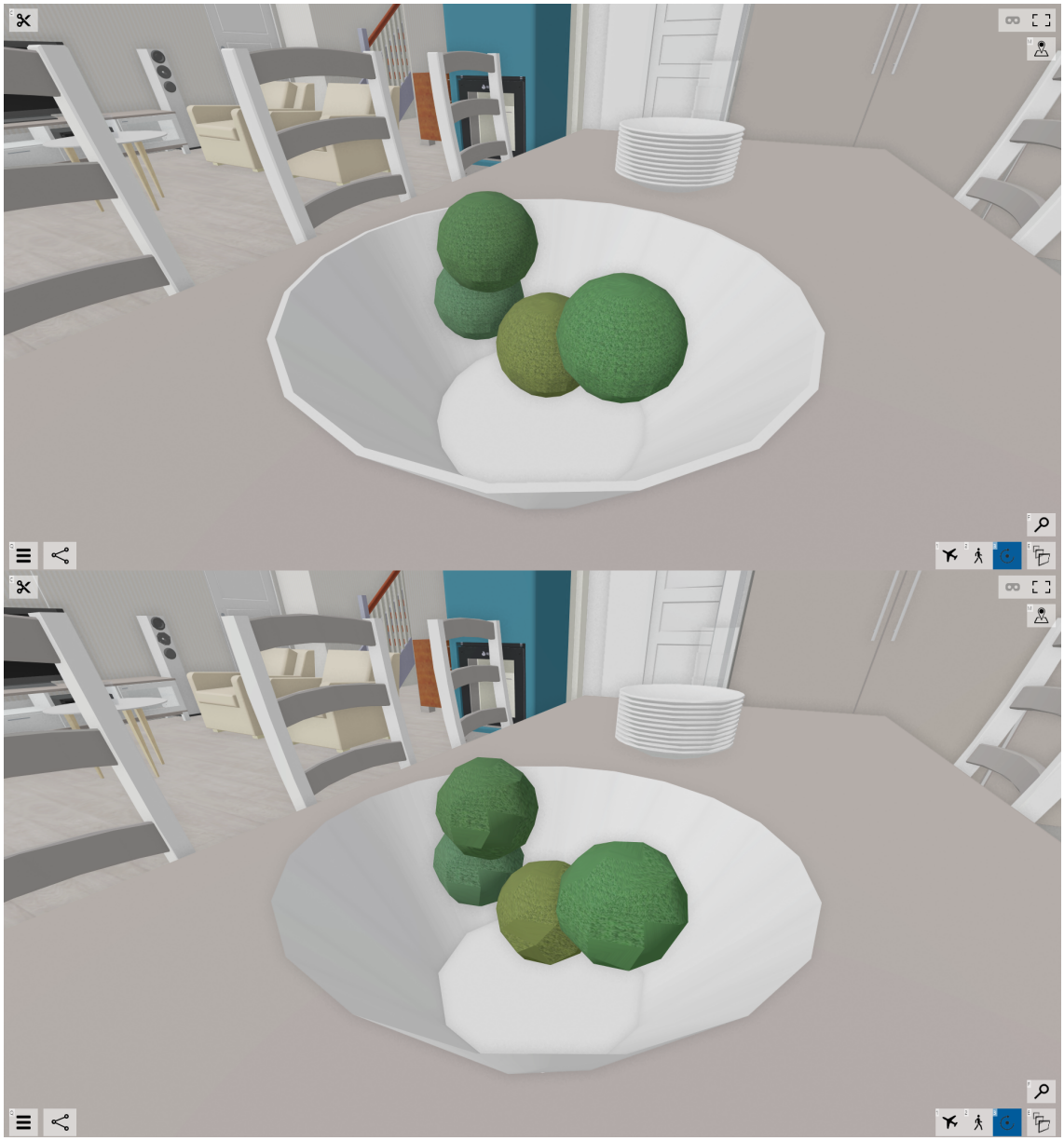


Figure B.2 *Original and simplified Vertex CAD building model. Round objects are simplified to a significant extent.*

Figure B.3 *Original and simplified Vertex CAD building model. Door handle which contains lots of high frequency detail is simplified, while all low frequency detail such as walls, doors and other decorative objects are in place.*

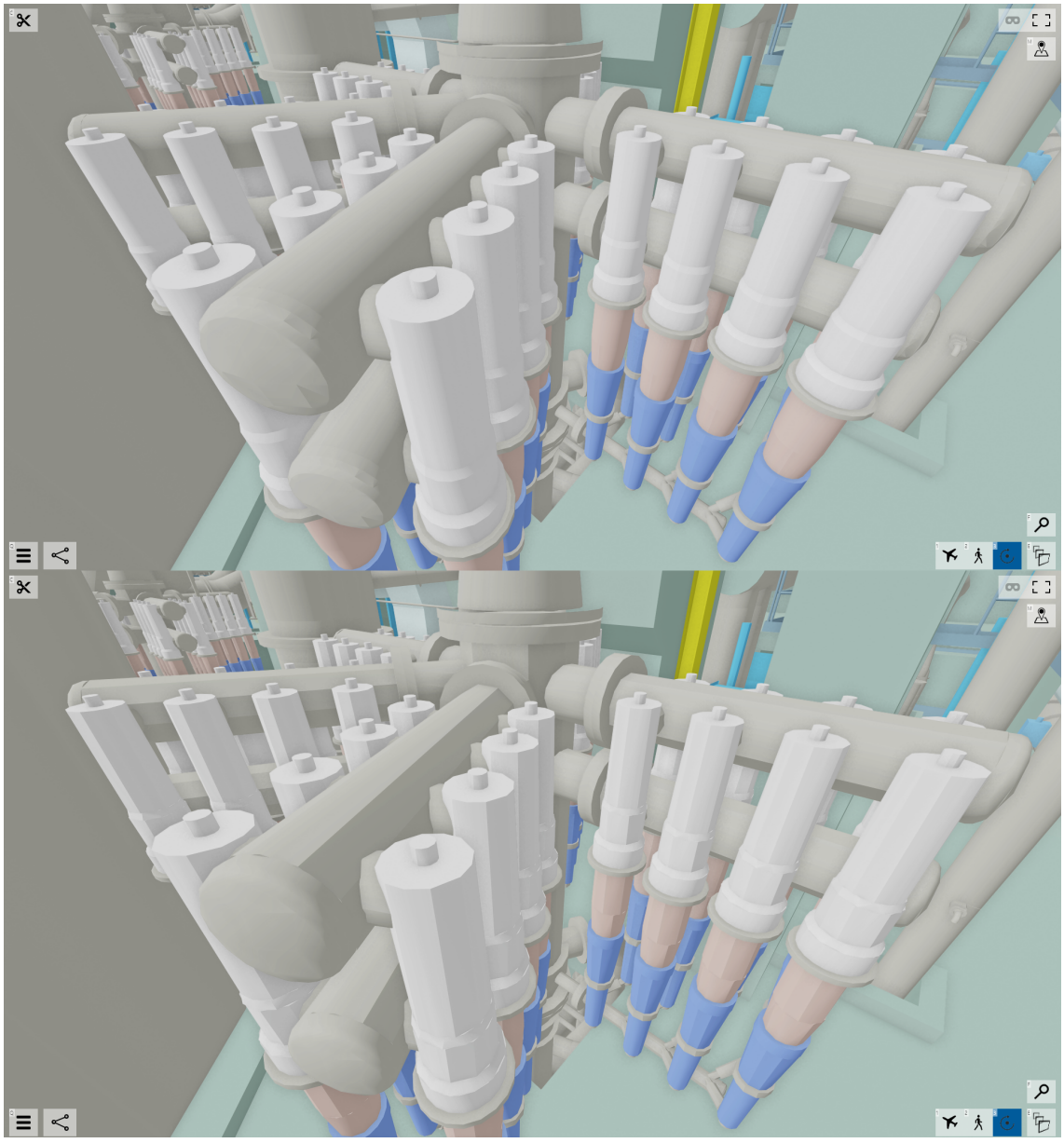


Figure B.4 *Original and simplified Vertex CAD plant model. Again, round elements benefit from the simplification, the overall shape is still remained while great amount of geometry has been decimated.*

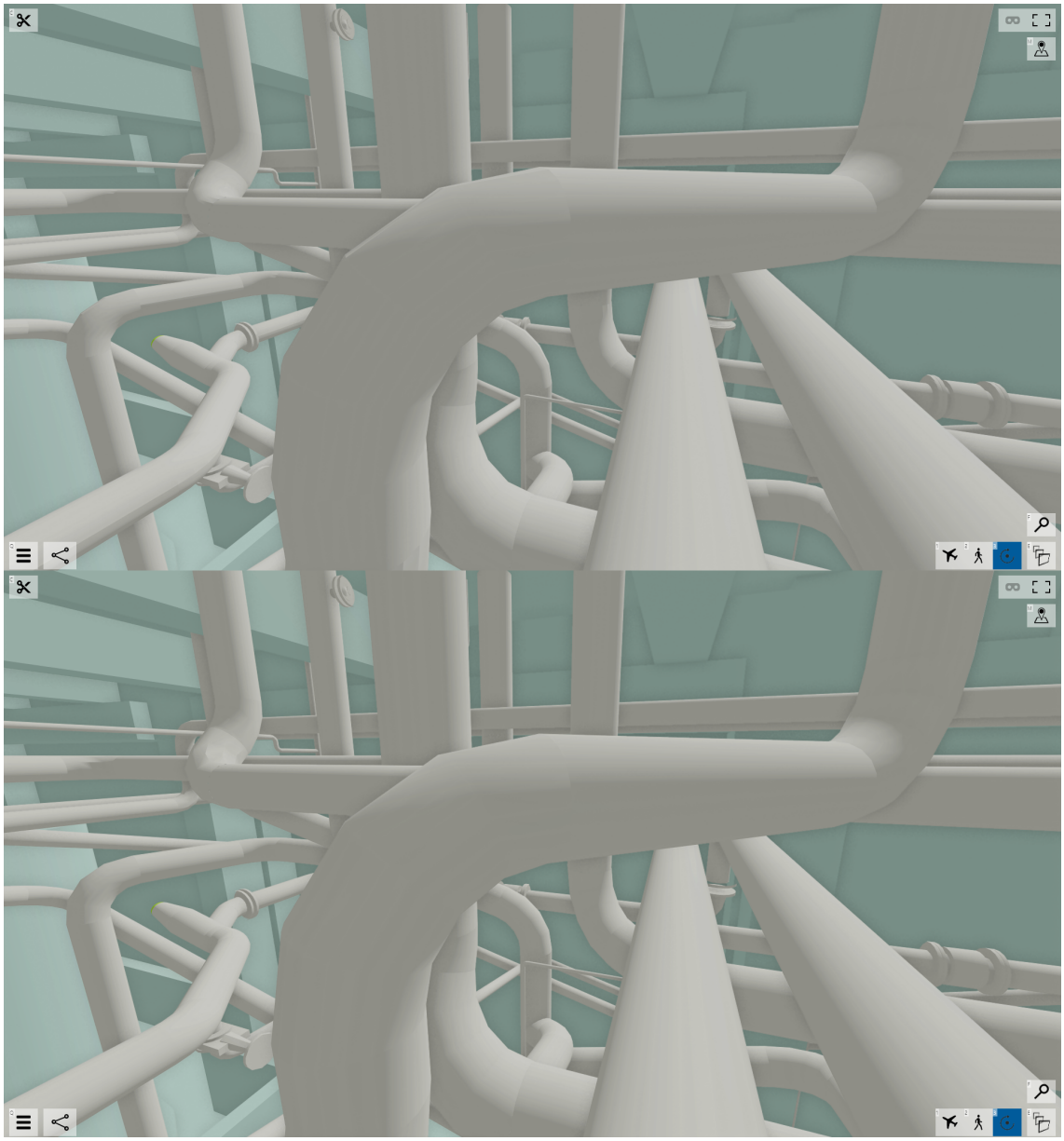


Figure B.5 *Original and simplified Vertex CAD plant model. In this specific plant model, piping mesh is from non-manifold surfaces and due to constraints, cannot be simplified.*