Linköping University | Department of Electrical Engineering
Master's thesis, 30 ECTS | Computer Graphics and Visualization
202017 | LIU-ISY/LITH-EX-A--2017/001--SE

Evaluation of an Appearance-Preserving Mesh Simplification Scheme for Configura AB

Rasmus Hedin <rashe877@student.liu.se>

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Contents

1 Introduction

For many years the field of computer graphics has been an important part in many industries, but especially in the entertainment industry (for instance video games and motion pictures). These industries generate a lot of money, and are quickly growing in size. A recent survey by *Kroon and Nilsson* [8] from *Dataspelsbranschen* have shown that the video games industry in Sweden generated €1325 M in revenue in 2016, a steep increase from the €392 M in 2012. Also, most movies nowadays use to some extent 3-D computer graphics in scenes where the cost would be to large to reproduce in reality, be too risky for actors, or simply be impossible.

The rendering of meshes (a collection of polygons describing a surface) is one of the main activities in computer graphics (usually, a collection of meshes; a so called scene description). In many cases, these meshes are very detailed, and require a large amount of polygons to fully describe a surface. This is problematic, since the rendering time of a scene depends on the number of polygons it has. Therefore, it is important to reduce the number of polygons in a mesh as much as possible. This is especially true in video games, where the scene needs to be rendered in real-time. However, if the number of polygons are reduced too much, it will degrade the visual quality of a mesh, giving a progressively flatter surface than intended and removing small surface details. This destroys the intended *geometrical appearance* of the mesh.

1.1 Motivation

While the geometrical appearance of a mesh is important, it is not the only factor which gives the final appearance of a mesh when rendering. According to *Cohen et al.* [1], both the surface curvature and color are equally as important contributors. *Textured appearance* will be used as the common name for these since surface properties are usually specified with a texture map.

In computer graphics, the process to reduce the number of polygons in a mesh based on some metric is called a *mesh simplification algorithm*, as seen in *Talton's survey* [16] in the field. Historically, these have been mostly concerned with minimizing the geometrical deviation of a mesh when applying it. Somewhat recently, methods for minimizing the texture deviation when simplifying a mesh have also appeared. They attempt to reduce the texture deviation and stretching caused when removing polygons from a mesh, as described in *Hoppe et al.* [5].

By simultaneously taking into account the geometrical and texture deviation, one can preserve the *visual appearance* of a mesh when simplifying it. If polygons can be removed without affecting this appearance significantly, the rendering time can be reduced for "free".

1.2 Aim

To survey the field for state-of-the-art mesh simplification algorithms that preserve the visual appearance of a mesh, and integrate these into *Configura's* (see Section 1.5) graphics pipeline. This will enable Configura to generate better *Level of Detail* (LoD) meshes for speeding up their rendering time. Currently, Configura only takes into account the geometrical deviation when simplifying, with no regard for the textures (e.g. diffuse or normal) on top of the mesh.

Thereafter, we plan to evaluate each of these solutions by measuring their performance and ability to preserve the meshes' original appearance. In the end, the goal is to find the mesh simplification algorithm which both performs and preserves the mesh appearance well.

1.3 Research Questions

After implementing and measuring the performance of these mesh simplification algorithms, answers to the questions below should have been obtained. These will be used to decide on a suitable alternative for Configura and also other systems with a similar set of requirements.

1. What alternative *mesh simplification schemes* exist that *preserves the appearance* of a mesh?
2. Which of these alternatives give the best *performance* and *appearance preservation*? When:
   a) Measuring the algorithm's *computation time* while targeting an *appearance threshold*?
   b) Measuring the algorithm's *memory usage* while targeting some *appearance threshold*?
   c) Measuring the *rendering time* of the *simplified mesh*? (by using the meshes generated according to the target *appearance threshold* above)
3. Which of the alternatives gives the best *appearance preservation* for a target *polygon count*?

1.4 Delimitations

Since there are many mesh simplification algorithms in previous work, a proper literature review would have to be done to find possible candidate solutions. Since our thesis' goals are mostly concerned with implementing and evaluating each solution, we have decided to base our choices on existing surveys and literature reviews to skip doing some of these ourselves.

Also, since implementing and doing measurements on all algorithms would take too long, we have decided to only pick an interesting subset of the algorithms presented in the surveys. More precisely, we have chosen to pick three different mesh simplification algorithms, one that does not take texture deviation into account and two that do. In addition, we will also compare them to Configura's existing mesh simplifier scheme; for a total of four algorithms.

1.5 Background

This thesis was requested by Configura AB, a company in Linköping which provides space planning software. Their main product, *CET designer*, lets companies plan, create and render 3-D office spaces (among other things). These scenes can have a large amount of polygons that need to be rendered in real-time for customers to evaluate their creations in CET designer.

To allow larger scenes to be rendered with higher frame-rates (e.g. needed when exploring environments in *Virtual Reality* (VR), to prevent motion sickness), it would be beneficial to reduce the amount of polygons as much as possible. The meshes in these scenes usually have textures applied to them, and it is therefore important to keep the quality as high as possible.

While Configura already has a mesh simplification in their pipeline, it only accounts for surface simplifications, and does not take into account the texture appearance that might be degraded when applying mesh simplification. Hence, the given task was to integrate a new mesh simplification scheme that takes into account texture quality when simplifying a mesh.

2 Theory

Since several mesh simplification algorithms are being considered, Section 2.4 presents the most notable schemes found (through peer-reviewed surveys) in previous work. An outline of the algorithm and the results found by authors are given for the reader's convenience, and also to be used as a guideline when implementing the solutions into Configura's CG pipeline.

Afterwards, in Section 2.5, we discuss the different metrics that can be used to measure the appearance preservation after a simplification has been done. This will later be used to evaluate the solutions and provide an empirical way to answer research questions 2 and 3 by giving a concrete metric for appearance thresholds and the amount of appearance deviation.

Finally, in Section 2.8, the methods and common practices for measuring performance of an algorithm are discussed. Based on existing industry practices, we show how to measure the computation time and memory usage of the algorithms. Since these measurements can be noisy, statistical methods will need to be used to truthfully answer our research questions.

2.1 Mesh Transformations

The detail of a mesh can be reduced/increased by altering the vertices and edges of the mesh. There exist several mechanisms that can be used.

- Edge collapsing
- Vertex removal
- Pair contraction
- Vertex split

In edge collapsing, the vertices of the edge $e_{ij} = (v_i, v_j)$ is removed and replaced with a single vertex v somewhere in between. The triangles that used the edge e_{ij} is removed and the neighbors of v_i, v_j will be connected with the new created vertex. This can be seen in figure 2.1(a).

Vertex removal is similar to edge collapsing, but now only one vertex is removed and the other vertex is moved to the location of the removed vertex. The neighbors of the removed vertex will be connected to the one that remains. In figure 2.1(b) vertex v_j of edge e_{ji} is removed and vertex v_i is moved to the location of v_j.

In a pair contraction, it is possible to merge two distant vertices that are not connected by an edge. Usually, the vertices have to within a certain threshold to be considered for a contraction. In figure 2.1(c) the vertices v_j, v_k is contracted into the new vertex v_i.

To increase the detail of the mesh an existing vertex can be split into two, i.e. vertex split. When performing a vertex split a vertex v is split into two new vertices v_i, v_j connected by an edge e_{ij}.

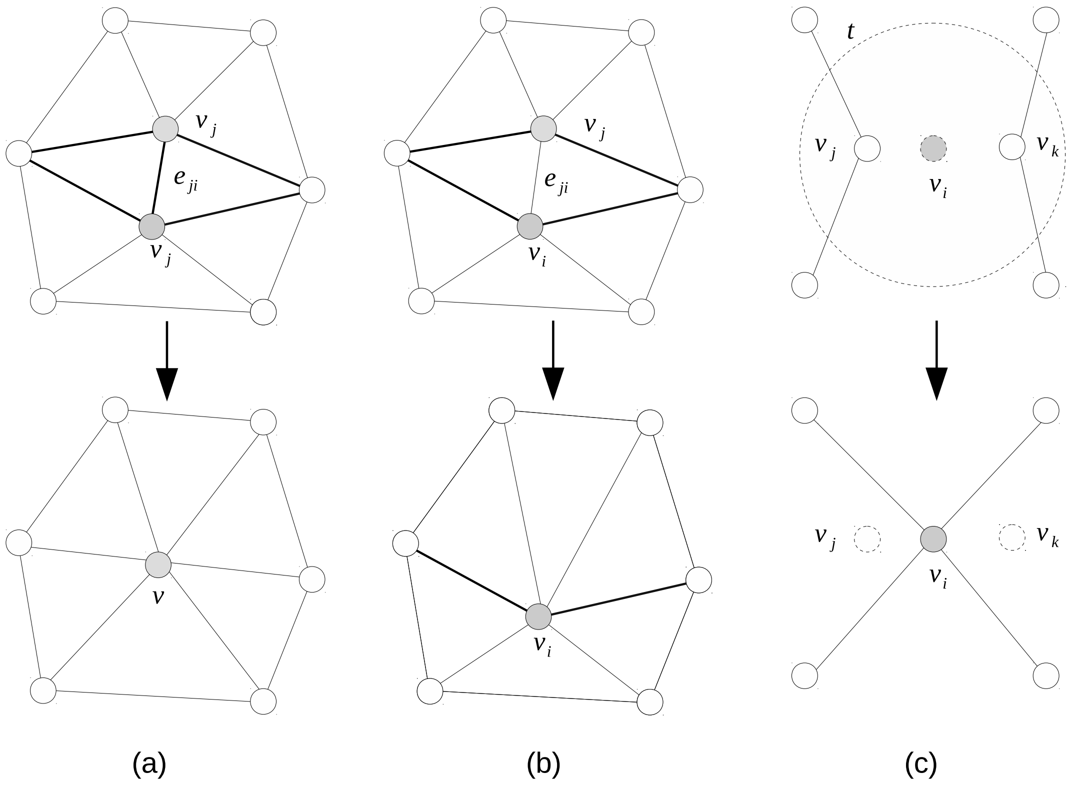

Figure 2.1: (a) edge collapse, (b) vertex removal, and (c) pair contraction

2.2 Mesh Parameterization

In order to apply, for example, a texture to a mesh each vertex is given a texture coordinate. This problem of finding a mapping between a surface (mesh)and a parameter domain (texture map) is called parameterization. Given a triangular mesh Hormann et al. [7] refers to the mapping problem as mesh parameterization. According to Hormann et al. there exists a one-to-one mapping between two surfaces with similar topology. Thus, a surface that is homeomorphic to a disk can be mapped onto a plane, i.e. a texture. If a mesh is not homeomorphic to a disk it have to be split into parts which are homeomorphic to a disk. These parts can then be put onto the same plane only one texture is needed.

2.3 Multiple attributes for a vertex

A vertex of mesh often have a property associated with it such as texture coordinates, color, and normal. A common way of texturing a model is to use a texture atlas where each triangle of the mesh is assigned a specific part of the texture. Since a texture is in a 2-dimensional domain a mesh parameterization needs to be performed. In many cases the mesh needs to be cut somewhere which will create seams. The vertices along this seam would need two sets of texture coordinates and this will create a discontinuity. In figure 2.2 a cylinder is unwrapped to 2D which creates a seam. All vertices along the seam will have texture coordinates (s, t) where $s = 0$ and $s = 1$.

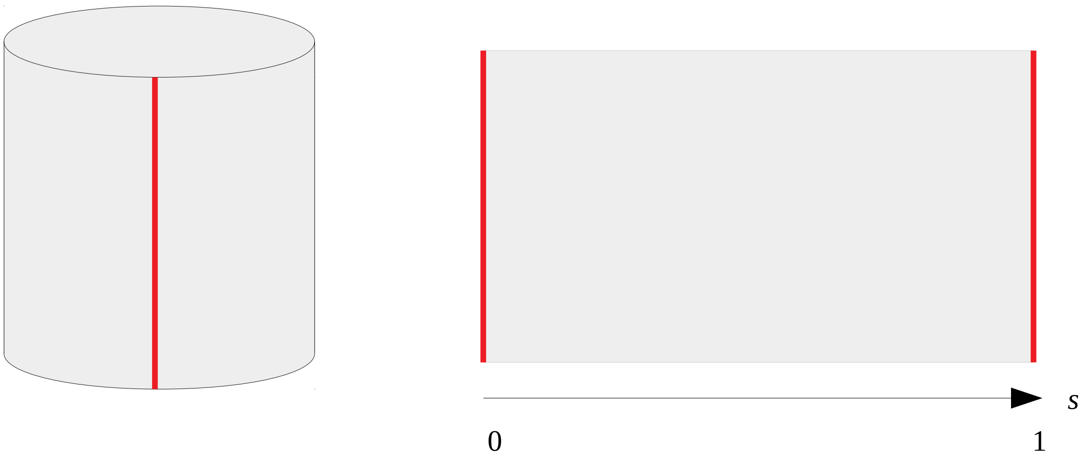

Figure 2.2: Unwrapping a cylinder. Vertices along seam (red line) need two texture coordinates

Hugues Hoppe [4] presents a mesh representation where each face adjacent to a vertex can have different appearnace attributes. The attribute values is associated with the corners of the faces instead of the vertices. A *corner* is defined by Hoppe as a tuple $< vertex, face >$. The attribute values at a corner defines the value that should be used for face f at vertex v. Corners with the same attributes and adjacent vertex are stored in a *wedge*. The wedge has one or more corners and each vertex will be divided into one or more wedges. This representation is useful for simplification of meshes with discontinuities in their attribute fields. Discontinuities could result in a simplified mesh where adjacent faces have been separated which introduces new holes in the mesh.

2.4 Mesh Simplification

According to *David Luebke's survey* [11] on the subject, mesh simplification is a technique which transforms a geometrically complex mesh (with a lot of polygons) to a simpler representation by removing unimportant geometrical details (reducing the number of polygons). It does this by assuming that some meshes are small, distant, or have areas which are unimportant to the visual quality of the rendered image. For example, if the camera in a scene always faces towards a certain direction, then removing details from the backside of a mesh won't affect the final rendered result, since they will never be seen by the camera anyway. Reducing the number of polygons allows meshes to use less storage space and need less computation time.

There are many mesh simplification algorithms, as can be seen in *David Luebke's survey* [11], each presenting a new approach with their own strengths and weaknesses. The first scheme is due to *Schroeder et al.* [15] in 1992, called *mesh decimation*. It was meant to be used to simplify meshes produced by the marching cubes algorithm, which usually gives unnecessary details. It works by making multiple passes through the meshes' vertices, and deleting vertices that do not destroy local topology and are within a given distance threshold when re-triangulated.

While the scheme above is simple and fast, it unfortunately does not give a geometrically optimal simplification. But by using *quadric error metrics*, which we discuss in section 2.4.1, it is possible to achieve such an optimal result. We then consider texture preserving simplifiers.

2.4.1 Quadric-Based Error Metric

Besides *decimation-based methods*, such as the aforementioned mesh decimation scheme, there exists another class of simplifiers based on *vertex-merging mechanisms*. According to *Luebke* [11], these work by iteratively collapsing a pair or vertices (v_j, v_k) into a single vertex v_i. This will also remove any polygons which were suspended by (v_j, v_k). The first collapse scheme is due to *Hoppe et al.* [6], which shows an *edge collapse* of $e_{ji} = (v_j, v_i) \rightarrow v_i$ as in Figure 2.3 (a). There exist other schemes, such as *pair contraction* in Figure 2.3 (b), but these don't tend to preserve the local topology of the original mesh, and instead focus on a more aggressive simplification.

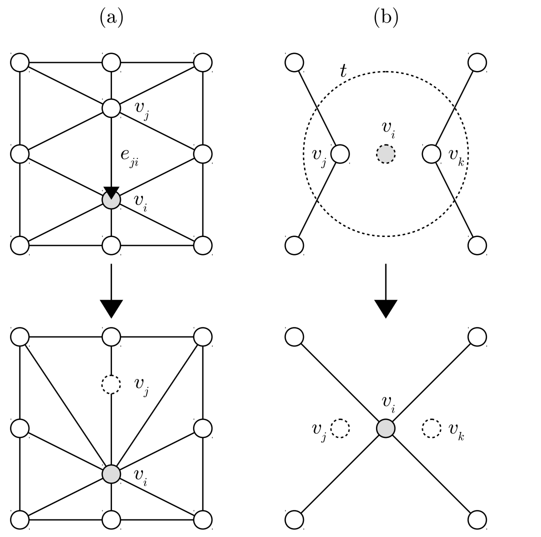

Figure 2.3: (a) edge $e_{ji} = (v_j, v_i)$ contraction towards v_i and (b) pair (v_j, v_k) contraction in the distance threshold t toward new vertex v_i.

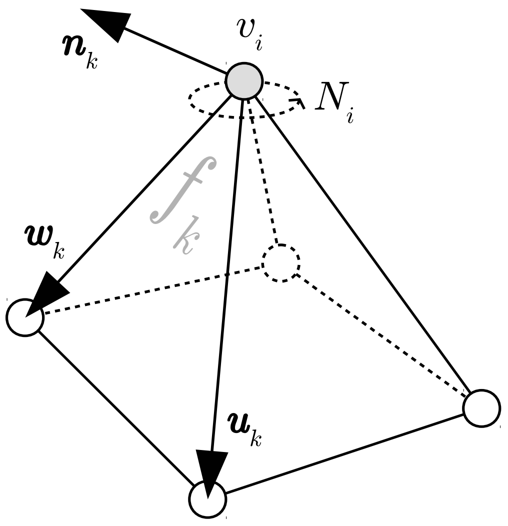

Figure 2.4: depiction of one of the planes f_k in the neighborhood N_i of the vertex v_i. It has a normal $\mathbf{n}_k$; found by the $\mathbf{w}_k \times \mathbf{u}_k$ of its edges.

Mesh simplification with *Quadric Error Metrics* (QEM), due to *Garland and Heckbert* [3], is based on the vertex merging paradigm. It provides a provably optimal simplification in each iteration, by collapsing the edge $(v_j, v_i) \rightarrow v_i$ and re-positioning it at $\bar{\mathbf{v}}$, which gives it the lowest possible geometrical error. By assigning a matrix $\mathbf{Q}_i$ for each vertex v_i, one can find the error $\Delta(\mathbf{v})$ introduced by moving v_i to $\mathbf{v}$. $\Delta(\mathbf{v})$ is the sum of distances from $\mathbf{v}$ to the planes $\mathbf{f}_k$ in v_i's neighborhood N_i (all polygons around v_i). Since $\Delta(\mathbf{v})$ is quadratic, finding a $\mathbf{v}$ which gives a minimal error is a linear problem. The best position $\bar{\mathbf{v}}_i$ for v_i after a collapse $(v_j, v_i) \rightarrow v_i$ is a solution to the eq. $(\mathbf{Q}_j + \mathbf{Q}_i)\bar{\mathbf{v}}_i = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix}^\top$. Thus, according to *Garland and Heckbert* [3]:

$$\bar{\mathbf{v}}_i = \begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{2.1}$$

$$\Delta(\mathbf{v}) = \mathbf{v}^\top \mathbf{Q}_i \mathbf{v}\ , \ \ \mathbf{Q}_i = \sum_{f_k \in N_i} \mathbf{f_k}\mathbf{f_k}^\top\ . \tag{2.2}$$

By storing the $\Delta(\bar{\mathbf{v}}_i)$ for every valid collapse $(v_j, v_i) \rightarrow v_i$ in a min-heap, the least cost collapse on the top of the heap can be done in each iteration, removing a vertex in each step. This is repeated until either a user-given vertex count $|\mathcal{V}|$ is reached or until some error threshold ϵ.

The results by *Garland and Heckbert* [3] show that QEM can reduce the simplification error by up to 50 % in comparison to a naïve scheme where $\bar{\mathbf{v}}_i = (v_i + v_j) \div 2$ and $\Delta(\mathbf{v}) = ||\mathbf{v} - v_i||$, as can be seen in Figure 2.5. They also argue that QEM gives higher-quality simplifications than *vertex clustering* and that it is faster than *progressive meshing* (which we also present later).

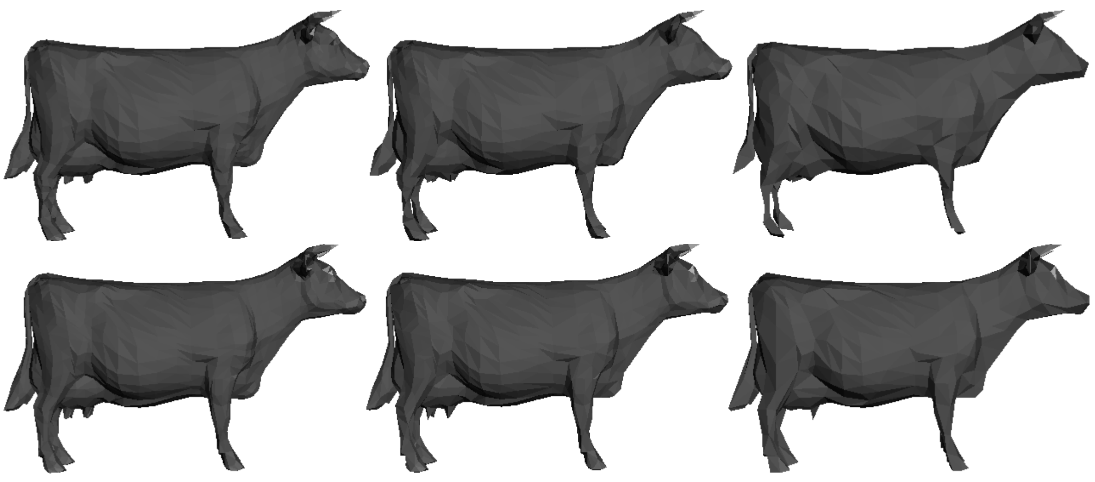

Figure 2.5: simplification using a naïve (top image) and a quadric error metric (bottom image) at different levels of detail at some vertex count (left to right: 50 %, 35 % and 17 % of original).

Garland and Heckbert [2] have also presented a more general QEM where the vertices can be placed in n-dimensional space. This makes it possible to, for example, include the color of the surface in the computation. Each vertex is treated as a vector $v \in R^n$. Thus, when the color is considered each vertex will be represented by a 6-dimensional vector $v = [x, y, z, r, g, b]^T$. The first three values is the spatial coordinates and the last three values will be the color. The same thing can be done with for example texture coordinates where each vertex would be represented by a 5-dimensional vector $v = [x, y, z, s, t]$ where s, t is the 2D texture coordinates.

The overhead of the higher dimensional quadrics is not extreme according to Garland and Heckbert. However, when using this with colors, normals, and texture coordinates some caution is needed. Colors may need to be clamped and normals needs to be normalized to unit length.

Garland and Heckbert have assumed that the properties vary continiously over the whole surface. However, if we want to for example apply a texture to a cylinder there will always be a seam where the ends of the texture meets. All vertices along this seam needs to be duplicated since they would require two different texture coordinates. The authors suggested having boundary constraints that would maintain the seam. However, in the case where a mesh have a corresponding texture atlas where each face have a specific part of the texture this might not work. The solution suggested by the authors is to allow multiple quadrics for each vertex, but, this is not yet implemented.

2.4.2 Progressive Meshes

Hugues Hoppe [5] introduced the *Progressive Mesh* (PM) representation as a scheme for storing and transmitting arbitrary polygon meshes. An abitrary mesh $\hat{M}$ in PM form is defined by a sequence of meshes $M^0, M^1, ..., M^n$ with increasing accuracy of the original mesh $\hat{M} = M^n$. Only the most coarse mesh M^0 is stored together with records of *vertex splits* that is used to refine M^0 into the more detailed meshes. A vertex split will transform M^i the more detailed mesh M^{i+1} and an edge collapse transforms M^i to the coarser mesh M^{i-1}.

To construct a PM the edge collapses of the original mesh needs to be determined. Multiple possible algorithims for chosing those edge collapses is presented by Hoppe [5], some with high speed but low accuracy and some more accurate but with lower speed. A fast but maybe not so accurate strategy is to choose the edge collapses at random but with some conditions. Another more accurate scheme is to use heursitics. According to Hoppe, the construction of a PM is supposed to be a preprocess, Therefore, the author chose an algorithm that take some time but is more accurate.

Hoppe based the simplification on the previous work *Mesh Optimizaton* [6] where the goal is to find a mesh that fits a sets X of points $x_i \in R^3$ with a small number of vertices. This problem is defined as a minimization of the energy function.

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M)n \tag{2.3}$$

Distance energy $E_{dist}(M)$ measures the sum of the squared distances from the points to the mesh. *Spring energy* E_{spring} is used to regularize the optimization problem. *Scalar energy* E_{scalar} measures the accuracy of the scalar attributes of the mesh. The last term, E_{disc} measures the geometric accuracy of the discontinuity curves (e.g creases). All legal edge collapses is placed in a priority queue where the edge collapse with lowest ΔE (estimated energy) is on the top of the queue. After a transformation is performed, the energy of the neighboring edges is updated.

2.4.3 Texture Mapped Progressive Meshing

Given an arbitrary mesh, *Sander et. al* [13] presents a method to construct a PM where a texture parametrization is shared between all meshes in the PM sequence. In order to create a texture mapping for a simplified mesh, the original mesh's attributes, e.g normals, is sampled. This method was developed with two goals taken into consideration:

- Minimize *texture stretch*: When a mesh is simplified the texture may be stretched in some areas which decrease the quality of the appearance. Since the texture parametrization determines the sampling density, a balanced parametrization is preferred over one that samples with different density in different areas. The balanced parametrization is obtained by minimizing the largest texture stretch over all points in the domain. No point in the domain will therefore be too stretched and thus making no point undersampled.

- Minimize *texture deviation*: Conventional methods use geometric error for the mesh simplification. According to the authors this is not appropriate when a mesh is textured. The stricter texture deviation error metric, where the geometric error is measured according to the parametrization, is more appropriate. This is the metric by *Cohen et al.* [1] explained in Section 2.4.4. By plotting a graph of the texture deviation vs the number of faces, the goal is to minimize the height of this graph.

Cohen et al. [1] stored an error bound for each vertex in a PM. *Sander et al.* [13] instead tries to find an atlas parametrization that minimizes both texture deviation and texture stretch for all meshes in the PM.

2.4.4 Appearance-Preserving Simplification

In order to preserve the appearance of a model when it is simplified, *Cohen et al.* [1] defines a new *texture deviation metric*. This metric takes three attributes into account: Surface position, surface curvature, and surface color. To properly sample these attributes from the input surface, the surface position is decoupled from the color and normals stored in texture and normal maps, respectively. The metric guarantees that the maps will not shift more than a user-specified number of pixels on the screen. This user-specified number is defined as ϵ.

Approximation of the surface position is done offline with simplification operations such as edge collapsing and vertex removals. At run-time, the color and normals are sampled in pixel-coordinates with mip-mapping techniques. Mip-maps are a pre-calculated sequence of images with progressively lower resolution. Here the decoupled representation is useful since the texture deviation metric can be used to bound how much the mapped attributes value's positions deviate from the positions of the original mesh. This guarantees that the sampling and mapping to screen-space of the attributes is done in an appropriate way.

Before any simplification can be made, a parametrization of the surface is required in order to store the color and normals in maps. If the input mesh does not have a parametrization, it is created and stored per-vertex in texture and normal maps. Next, the surface and maps are fed into the simplification algorithm which decides which simplification operations to use and in what order. The deviation caused by each operation is measured with the texture deviation metric. A *progressive mesh* (PM) with error bounds for each operation is returned by the algorithm, which can then be used to create a set of LOD with error bounds. Using the error bounds, the tessellation of the model can be adjusted to meet the user-specified error ϵ.

2.5 Metrics for Appearance Preservation

Previously in sections 2.4.4 and 2.4.3, the metrics texture deviation and texture stretch have been defined. But to measure more exactly how much the visual appearance of a simplified mesh deviate from the original mesh another metric would be better. *Lindstrom and Turk* [10] defines *image-driven simplification* which captures images from different angles of the mesh. The difference between the images of the original and simplified mesh are computed in order to measure how well the appearance is preserved. This metric is more general and can be applied to all simplification algorithms since it only compares the original mesh to the simplified mesh.

The *image metric* is defined as a function taking two images and gives the distance between them. To measure the distance the authors use root mean square of the luminance values of two images Y^0 and Y^1 with dimensions $m \times n$ pixels. It is defined as:

$$d_{RMS}(Y^0, Y^1) = \sqrt{\frac{1}{mn}\sum_{i=1}^{m}\sum_{j=1}^{n}(y^0_{ij} - y^1_{ij})^2} \tag{2.4}$$

To evaluate the quality of the simplified mesh the authors capture images from 24 different camera positions. The positions are defined as the vertices of a rhombicuboctahedron which can be seen in figure 2.6. Two sets of l images $Y^0 = Y^0_h$ and $Y^1 = Y^1_h$ with dimensions $m \times n$ is rendered and the RMS is then computed as:

$$d_{RMS}(Y^0, Y^1) = \sqrt{\frac{1}{lmn}\sum_{h=1}^{l}\sum_{i=1}^{m}\sum_{j=1}^{n}(y^0_{hij} - y^1_{hij})^2} \tag{2.5}$$

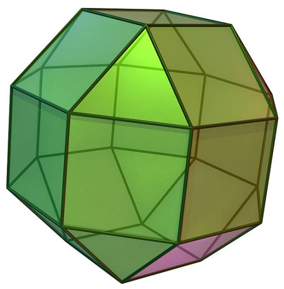

Figure 2.6: Rhombicuboctahedron with 24 vertices which is used as the camera positions. (Rhombicuboctahedron by Hellisp / CC BY 3.0)

2.6 Ray Casting

2.7 Octree

2.8 Measuring Algorithmic Performance

According to *David Lilja* [9, p. 4], there are three fundamental techniques that can be used when confronted with a performance-analysis problem: *measurement*, *simulation* or *modeling.* While the book concentrates on evaluating computer performance, these techniques can also be applied when evaluating different algorithms. Measurement would be to actually execute the implemented algorithm and simultaneously gather interesting statistics (e.g. how long it took to finish and how much memory was needed), and use this to compare the algorithms. While modeling would be to analytically derive an abstract model for the algorithm (e.g. the Big $\mathcal{O}$ worst-case running time and memory), and see which of them has a lower complexity.

Since not all of the algorithms in Section 2.4 have an analytical model derived by the authors, and also because the algorithms are to be evaluated in a real system, only the problems inherent to the measurement approach will be considered. One of the problems with doing measurements of a real system (a program running on a computer in this case), according to *David Lilja* [9, p. 43], is that they introduce noise. This noise needs to be modeled to be able to reach a correct conclusion, such as determining if algorithm A is faster than algorithm B. One way of doing this, according to *David Lilja* [9, p. 48], is to find the confidence interval of the measured value, by assuming the source's error is distributed according to some statistical distribution (like the Gaussian or the Student t-distribution). The confidence interval $[a, b]$ when assuming the source's error is t-distributed, can be found as shown below. Where n tests are taken (giving $n - 1$ degrees of freedom), with a significance level of α (usually 5 %).

$$a = \bar{x} - t_k \frac{s}{\sqrt{n}}, \quad b = \bar{x} + t_k \frac{s}{\sqrt{n}}, \quad t_k = t_{1-\alpha/2,n-1} \tag{2.6}$$

One common mistake, according to *Schenker et al.* [14], when using confidence intervals to determine if e.g. an implemented algorithm A is faster than B, is the use of the overlapping method to reach conclusions. If two confidence intervals *do not* overlap, then the result is provably significant (that is, algorithm A is either faster or slower than B). However, the converse is not true, if two intervals *do* overlap, then no conclusions can be reached since the result could be either significant or not significant.

3 Method

Now that the theoretical groundwork has been laid out, we describe in Section 3.2 how the solution was implemented into Configura's graphics pipeline and then show how to evaluate it in Section 3.3 according to the posed research questions. In more detail, the implementation description shows how the different algorithms are implemented in practice and how these fit into Configura's pipeline. Moreover, we also show how the appearance evaluator has been implemented and integrated into the system, which provides a uniform way to simplify a mesh until a certain appearance threshold has been reached. In the evaluation part of the method, we describe how to measure the computation time, memory usage, polygon count and appearance preservation of a algorithm given a certain mesh and parameters. This method will thereafter be used to acquire the results needed to answer our research questions.

3.1 System Overview

Before describing the system in detail, an overview is given in the Figure 3.1 below. In brief, the original mesh that is going to be simplified is given as input to the simplification algorithm, and as output comes a simpler mesh with less polygons. In order to provide a common interface to each of the algorithm's parameters, an appearance evaluator takes as input an appearance threshold (given as a RMS value) and the original and candidate meshes. This will provide the specific parameters to the algorithm in question, and incrementally reduce the quality measure (e.g. $|\mathcal{V}|$ or ϵ) until the appearance threshold of a mesh candidate is satisfied.

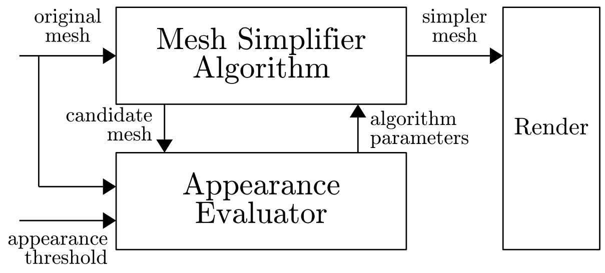

Figure 3.1: System Overview of a Simplifier Pipeline

3.2 Implementation

Before the chosen mesh simplification algorithm can be integrated into Configura's CET Designer, the three aforementioned different algorithms need to be implemented and evaluated. Only then can we make an informed decision on which of the algorithms are most suited for Configura, and thereafter integrate that solution into the CET Designer. In this section we describe how the three algorithms were implemented and the tools used to accomplish it. **Note:** for the course TDDD89 only a brief and preliminary overview is given over the implementation, since this part is best written after we've actually implemented some algorithms.

The first step is to decide how and where to implement the simplification algorithms and the evaluation system. One possibility is to implement them directly in Configura's own language *CM* (built in-house, a mix between C and Lisp). However, because we first only want to evaluate the algorithms without the overhead that may be introduced by CET Designer, this might not be the best idea. Therefore, we have decided to implement and evaluate the algorithms outside of CET Designer by building a simple testbed using C/C++ and the *OpenGL* API (for evaluating the appearance and also to measure its rendering speed).

To be able to change between the different algorithms, a common interface is implemented that can interact with the appearance evaluator. As can be seen in Figure 3.1, the simplification algorithm takes a mesh and then gives a simplified mesh to the evaluator which will compare it to the original mesh. The simplification will be performed until a given appearance-threshold is reached (thus, giving all of the algorithms a universal stopping condition).

A quick overview of how the algorithms work is explained in the theory chapter, and since the actual implementation is based on this description, only small details should deviate from the theory. But in order to give a more detailed description of how they are implemented, the practical details will also be described in the coming sections. However, as mentioned before, these sections will be mostly left empty at the moment since the implementations is not done yet for the scientific methods course. We then describe how to integrate the chosen solution.

3.2.1 Quadric-Based Error Metric

As a rule, the original mesh will be given as an ordinary triangle mesh (a so called “triangle soup”), which is not suitable for applying the QEM algorithm (since the local neighborhood information isn't available). Instead, we convert this triangle soup to a half-edge mesh. This allows easy manipulation of the local neighborhood of the mesh, which is precisely what is needed when doing a edge collapse or when calculating the error quadrics of a given vertex.

After doing this, the implementation basically follows the theoretical framework to the letter, where the least-cost edge is chosen to be contracted from the min-heap. Lastly, this edge is collapsed and then the remaining “hole” is simply (but with a few special cases...) linked back together so that the local neighborhood of the vertex still qualifies as a closed manifold. All source code is provided at the end of this report (**not** for the TDDD89 course).

3.2.2 Appearance-Preserving Simplification

3.2.3 Texture Mapped Progressive Meshing

3.2.4 Integrating Solution into the Pipeline

When the candidate algorithm has been chosen, it has to be integrated into CET Designer. Since we've chosen to evaluate the solution in C/C++ in our own renderer, the algorithm needs to be ported to the Configura CM language. It also needs to interact with the existing file format of the models, which means that either an existing Configura library will be needed or a custom one will need to be written for our purposes. After that, the integration is mostly painless, because the algorithm doesn't depend that much on any of the other parts in CET Designer. It should just be a case of: *original mesh → mesh simplifier → simplified mesh.*

3.3 Evaluation

In order to determine which of these algorithms provide the best performance for a target appearance threshold, an evaluation of the polygon count, computation time, memory usage and rendering time of the simplified mesh is done for each of the implemented solutions. In the results from this step, a series of tables are generated to compare the performance between the algorithms by using a common comparison framework. In this section, we describe this common comparison framework and then show how we can measure each of the parameters.

In essence, this is done by targeting a certain appearance threshold, tweaking the mesh simplification algorithm's parameters to achieve this threshold, and then measuring the given performance. This gives a universal measure of "quality" for all of the algorithms, which would otherwise have different error metrics used for applying the simplification. Since the performance measures are noisy, a total of $n = 20$ samples will be taken. According to *David Lilja* [9, p. 50] the t-student distribution should be used when $n < 30$, as shown in Section 2.8.

The pack of test meshes that are going to be used in the comparison are a combination of textured models provided by Configura and others taken from the public domain. The exact selection of these is still to be decided, but should include both low- & high-polygon meshes.

3.3.1 Appearance Preservation

In order to compare the appearance preservation of the mesh simplification algorithms, the image-metric explained in section 2.5 is used. It is useful since it can compare the difference of any two meshes, therefore, it does not depend on the algorithm used.

For both the original mesh and a simplified mesh, 24 images with resolution 512×512 is rendered with a simple renderer based on *OpenGL*. The camera is placed at the vertices of a rhombicuboctahedron and is faced towards the center where the mesh is placed. A light source is placed at the camera position. This will make sure that the surface facing the camera will be illuminated.

The two sets of 24 images each is used to compute the RMS of the simplified mesh with equation 2.5. This RMS value can then be used to compare how well the algorithms perform.

3.3.2 Polygon Count

Concerning research question 3, the appearance preservation for a specific target polygon count needs to be measured. Therefore, the simplification algorithms is tasked to simplify until the target polygon count is reached. When it is reached, the image-metric is used to measure how well the appearance is preserved. Measurements will be performed for multiple target polygon counts.

3.3.3 Computation Time

An important property of a mesh simplification algorithm is the time it takes to simplify a full-resolution mesh to a lower-resolution mesh. While this doesn't impact the run-time of the mesh (that is accounted for by the rendering time, measured in Section 3.3.5), it is still important to reduce it as much as possible. This is especially true if the LoD is dynamically generated at run-time, but it is also important since many more meshes can be simplified per time unit (important if a simplifier is to be provided as a service, as *Simplygon Linköping* does).

In order to compare the execution time of the different algorithms, they all target the same appearance thresholds as specified in Section 3.3.1 by tweaking the parameters unique to each algorithm. We then measure the time it takes for the simplification algorithm to execute when using these parameters, in other words, simply by: $time_{\mathcal{A}} = end_{\mathcal{A}} - start_{\mathcal{A}}$ for an algorithm $\mathcal{A}$. Of course, this measurement is done several times to account for noise. After calculating the mean $\bar{x}$ and the standard-deviation s, one can find the confidence interval $[a, b]$ of the execution time by the equations shown in Section 2.8 with 19 degrees of freedom and $\alpha = 5\%$.

3.3.4 Memory Usage

Another important performance property of the algorithm is the accumulated memory used when simplifying the mesh. Depending on the size of the mesh in triangles, the algorithm could consume large amounts of memory (and might not even fit in the primary memory in some cases). It is therefore important to compare the simplification algorithms to determine those which are suited for optimizing large triangle meshes and those that aren't. Since this measure will always be deterministic (at least for the method we use to measure it), there is no need to apply any statistical measures. The *Valgrind* suite was chosen since it has the *Massif* heap profiler, which gives accurate memory usage. According to the *Massif documentation* [12] there is an expected slowdown of 20x, which isn't a problem since the computation time and rendering time are measured separately. Below are the commands to find the memory usage.

```
valgrind --tool=massif ./simplify –algorithm=<algorithm> <input-mesh> <output-mesh>
```

3.3.5 Rendering Time

One main purpose of simplifying a mesh is to reduce the rendering time. Therefore, the frame rate is measured when rendering the original mesh and the simplified meshes. The simplified meshes comes from the previous steps where different appearance-thresholds was targeted. Frame rate is measured by counting how many frames that have been rendered during a period of one second. As with computation time, this measurement may have noise and therefore multiple samples needs to be obtained. The mean and confidence interval is obtained in the same way.

4 Results

5 Discussion

6 Conclusion

Bibliography

[1] Jonathan Cohen, Marc Olano, and Dinesh Manocha. “Appearance-preserving simplification”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 115–122.

[2] Michael Garland and Paul S Heckbert. “Simplifying surfaces with color and texture using quadric error metrics”. In: *Visualization’98. Proceedings*. IEEE. 1998, pp. 263–269.

[3] Michael Garland and Paul S Heckbert. “Surface simplification using quadric error metrics”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1997, pp. 209–216.

[4] Hugues Hoppe. “Efficient implementation of progressive meshes”. In: *Computers & Graphics* 22.1 (1998), pp. 27–36.

[5] Hugues Hoppe. “Progressive meshes”. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 99–108.

[6] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. “Mesh optimization”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM. 1993, pp. 19–26.

[7] Kai Hormann, Bruno Lévy, and Alla Sheffer. “Mesh parameterization: Theory and practice”. In: (2007).

[8] Jacob Kroon and Anna Nilsson. “Game developer index 2017”. In: *Based on 2017 Annual Reports. Dataspelbranchen* (2017).

[9] David J Lilja. *Measuring computer performance: a practitioner’s guide*. Cambridge University Press, 2005.

[10] Peter Lindstrom and Greg Turk. “Image-driven simplification”. In: *ACM Transactions on Graphics (ToG)* 19.3 (2000), pp. 204–241.

[11] David P Luebke. “A developer’s survey of polygonal simplification algorithms”. In: *IEEE Computer Graphics and Applications* 21.3 (2001), pp. 24–35.

[12] *Massif: a heap profiler*. `http://valgrind.org/docs/manual/ms-manual.html`. Accessed: 2017-12-14.

[13] Pedro V Sander, John Snyder, Steven J Gortler, and Hugues Hoppe. “Texture mapping progressive meshes”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 409–416.

[14] Nathaniel Schenker and Jane F Gentleman. “On judging the significance of differences by examining the overlap between confidence intervals”. In: *The American Statistician* 55.3 (2001), pp. 182–186.

[15] William J Schroeder, Jonathan A Zarge, and William E Lorensen. “Decimation of triangle meshes”. In: *ACM Siggraph Computer Graphics*. Vol. 26. 2. ACM. 1992, pp. 65–70.

[16] Jerry O Talton. “A short survey of mesh simplification algorithms”. In: *University of Illinois at Urbana-Champaign* (2004).