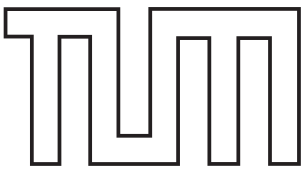


Technische Universität München
Chair of Media Technology
Prof. Dr.-Ing. Eckehard Steinbach

Interdisciplinary Project

Parallel Mesh Simplification with Adaptive Thresholding Based on Quadric Error Metrics

Author: Wojciech Zielonka
Address: Munich
Advisor: Prof. Dr.-Ing. Eckehard Steinbach
Begin: 01.04.2019
End: 01.01.2020

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

München, May 20, 2020 _____

Place, Date Signature

München, May 20, 2020 _____

Place, Date Signature

Abstract

This work elaborates a new parallel algorithm based on quadric error metric and adaptive thresholding to simplify a triangle mesh. The approach emphasizes planar surfaces as a target to simplify. The main goal was to create a framework able to produce high quality progressive meshes for browser streaming purposes.

Contents

Chapter 1

Introduction

Mesh simplification is necessary when one wants to reduce the size of a mesh while still preserving geometry. The technique is widely used in computer graphics to change the model level of details [LRC+03]. This project elaborates a specific case of mesh simplification, where the focus is mostly on planar surfaces, like walls and floors, at the same time, keeping high level of details for complex shapes; plants, elements on desks in an office, etc. Mesh reconstruction in general introduces a problem of using the same level of details for the whole $3D$ space. Most of planar surfaces can be described with reduced amount of triangles. Therefore, after generating a reconstructed mesh from real-world environments, mapped by static or mobile scanners, we can successfully apply simplification with great results. In the next chapter, I will elaborate foundations for the geometric error metric. Forwarded by the introduction to the extended version of this algorithm, which additionally uses color and normals for the error metric. Finally, in the forth chapter, I will describe the parallel approach with adaptive thresholding to solve the problem.

Chapter 2

Basic Simplification Algorithm

In this section, a basic algorithm will be presented for mesh simplification. The algorithm is founded on several fundamental components: iterative vertex contraction and quadric error metric. This part elaborates basics of one of the most popular methods for mesh simplification created by Michael Garland.

2.1 Design

The core of the algorithm is based on Michael Garland's work Quadric-Based Polygonal Surface Simplification [Gar99], where he suggests an algorithm capable of producing high-quality approximations of polygonal meshes. The main assumption is that the approximation need not to maintain the topology of the original surface and is a nicely balanced trade-off between quality and size.

The goal of this work was to adopt this algorithm to a parallel framework with adaptive thresholding, capable of fast progresive mesh streaming [SY01] for renderer engines in browsers. An example of such a renderer is Indoorviewer product created by NavVis. Depending on selected mesh resolution and level of details an appropriate mesh will be streamed to a browser. Therefore, the size and quality is crucial for the endpoint users to get maximal usability. Using the assumption that planar surfaces need much less triangles to describe geometry, while preserving details at non-planar surface. Light and detailed meshes can be generated which are suitable for streaming purposes.

2.2 Iterative Vertex Contraction

The simplification algorithm is based on several atomic operations. The most important of them is an edge contraction. A pair of contraction is defined as $(\mathbf{v_i}, \mathbf{v_j}) \rightarrow \bar{\mathbf{v}}$. The atomic operation of a contraction is then defined as:

1. Move the vertices $\mathbf{v_i}$ and $\mathbf{v_j}$ to the position $\bar{\mathbf{v}}$
2. Replace all connections of $\mathbf{v_j}$ with $\mathbf{v_i}$
3. Remove $\mathbf{v_j}$ and all faces which belong both to $\mathbf{v_i}$ and $\mathbf{v_j}$. In Figure 2.1 the gray faces.

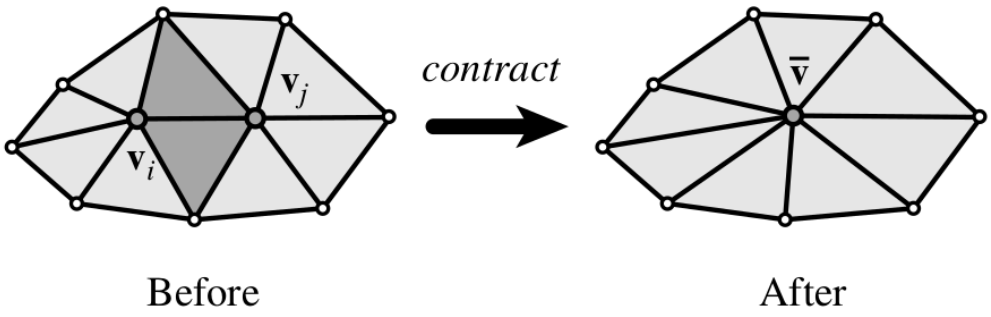


Figure 2.1: Contraction of an edge. Remove $\mathbf{v_j}$ and move remaining edges to $\mathbf{v_i}$ [Gar99].

The mentioned algorithm is a greedy procedure driven by the cost of contraction [CLRS01]. It stops when the current threshold level is reached. To achieve simplification we apply a sequence of edge contraction. Where the sequence is created as follows [GH97]:

1. Select a set of candidate vertex pairs.
2. Assign a cost of contraction to each candidate.
3. Place all candidates in a heap keyed on cost with the minimum cost pair at the top.
4. Repeat until the desired approximation is reached:
   (a) Remove the pair $(\mathbf{v_i}, \mathbf{v_j})$ of least cost from the heap.
   (b) Contract this pair.
   (c) Update costs of all candidate pairs involving $\mathbf{v_i}$.

Each edge is associated with a cost of contraction, which is basically the amount of error made during deletion of a given pair of vertices. This cost is a key in the minimum

heap [CLRS01] which is iteratively $pop()$. In each main iteration (steps from 1 to 4) we contract edges up to the current adaptive threshold level. If a current edge's cost is bigger or equal than the acceptable error, the main iteration procedure is stopped and the remaining edges in the heap are ignored. In a next iteration, the heap is rebuilt and the error level is slightly increased in such a way that previously contracted regions are even more simplified. The error calculations are based on a hyper-parameters *aggressiveness* and a current iteration value:

$$error(i) = 0.000000001 * (i + 3)^a \tag{2.2.1}$$

where i is the iteration and a is the aggressiveness.

The formula 2.2.1 is inspired by a Github implementation of Fast-Quadric-Mesh-Simplification, where the author introduced adaptive thresholding using 2.2.1. The *aggressiveness* has to be changed once the different error metric is used. The best results for geometry gives value 3 and for the rest of attributes 5 is used.

Rebuilding heap captures the change of geometry made in a contraction for the whole mesh. After one contraction we update only first order neighbors of a given vertex. Therefore, we need to rebuild heap to reflect the global change.

2.3 Assessing Cost of Contraction

In this section, the way how to measure the cost of contraction for an edge will be elaborated. For simplicity, the analysis is made just for geometry attributes. Maintaining high level of details and faithful representation of the original mesh, the cost should be reflected in the effect of changing geometry of the surface. Meaning, if the error is small, the geometry changes insignificantly. An edge with a small error is a good candidate for removal.

Because the metric is plane-based, the standard representation of a plane is defined as $\mathbf{n}^T\mathbf{v} + d = 0$ with normal $\mathbf{n} = [a\ b\ c]^T$, d is a scalar constant and $\mathbf{v} = [x\ y\ z]^T$ is a point in $3D$ space. From this, the quadric error metric can formulate as following [Gar99]:

$$D^2(\mathbf{v}) = (\mathbf{n}^T\mathbf{v} + d)^2 = (ax + by + cz + d)^2 \tag{2.3.1}$$

The error for the set of planes associated with the vertex v is then defined as (we have to remember that this set is purely conceptual):

$$\sum_i D_i^2(\mathbf{v}) = \sum_i (\mathbf{n_i}^T\mathbf{v} + d_i)^2 \tag{2.3.2}$$

Each vertex has an accumulated error metric value for surrounding faces which represents the maximum squared distance to the intersection of all planes spanned by each face.

Figure 2.2 shows how removing a pair of contraction $(\mathbf{v}_i, \mathbf{v}_j)$ could look in practice in the $2D$ case. Vertices $\mathbf{v}_i, \mathbf{v}_j$ define set of lines $P_i = \{A, C\}$ and $P_j = \{B, C\}$. The error on each of those vertices equals $E_{plane}(v_i) = E_{plane}(v_j) = 0$ since they both lie on the lines span by thier sets. Let me define a new set which is a union of $\bar{P} = P_i \cup P_j = \{A, B, C\}$. Position of $\bar{\mathbf{v}}$ minimizes the sum of square distances the the lines in $\bar{P}$ [Gar99].

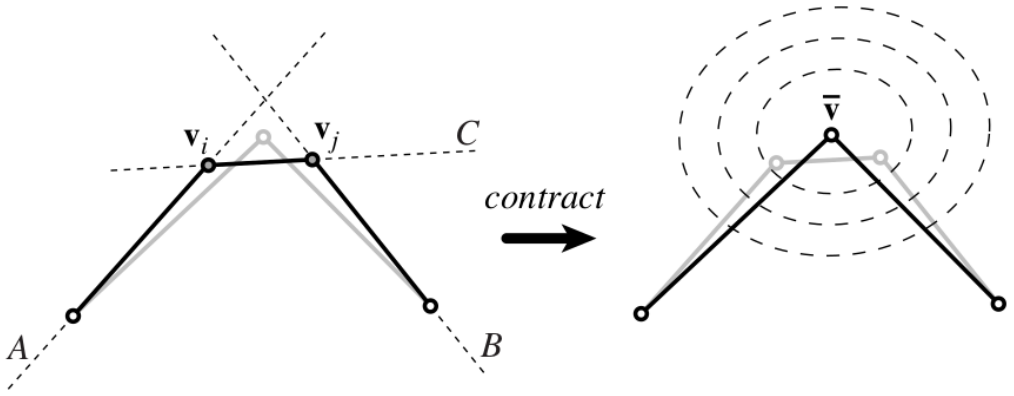


Figure 2.2: Measuring contraction cost in 2D where $\bar{\mathbf{v}}$ is a global minimum of our optimization objective [Gar99].

2.4 Quadric Error Metric

In this section the compact representation of the quadric error is introduced. First, previously declared formula of quadric distance is expanded to: $D^2(\mathbf{v})$ 2.3.1.

$$\begin{aligned} D^2(\mathbf{v}) &= (\mathbf{n}^T\mathbf{v} + d)^2 && (2.4.1)\\ &= (\mathbf{n}^T\mathbf{v} + d)(\mathbf{n}^T\mathbf{v} + d) && (2.4.2)\\ &= (\mathbf{v}^T\mathbf{n}\mathbf{n}^T\mathbf{v} + 2d\mathbf{n}^T\mathbf{v} + d^2) && (2.4.3)\\ &= (\mathbf{v}^T(\mathbf{n}\mathbf{n}^T)\mathbf{v} + 2(d\mathbf{n})^T\mathbf{v} + d^2) && (2.4.4) \end{aligned}$$

where $\mathbf{n}\mathbf{n}^T$ is the outer product of the face normal:

$$\begin{bmatrix} a^2 & ab & ac \\ ab & b^2 & bc \\ ac & bc & c^2 \end{bmatrix} \qquad (2.4.5)$$

Therefore, the *quadric* Q is defined as a triple:

$$Q = (\mathbf{A}, \mathbf{b}, c) \qquad (2.4.6)$$

Where **A** is a $3x3$ matrix, **b** is a 3-vector and c is a scalar. Therefore, the equation 2.4.4 is rewritten to:

$$Q(\mathbf{v}) = \mathbf{v}^T\mathbf{A}\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c \tag{2.4.7}$$

Quadrics provide an intuitive addition operation which is component-wise: $Q_i(\mathbf{v}) + Q_j(\mathbf{v}) = (Q_i + Q_j)(\mathbf{v})$ where $Q_i(\mathbf{v}) + Q_j(\mathbf{v}) = (\mathbf{A_i} + \mathbf{A_j}, \mathbf{b_i} + \mathbf{b_j}, c_i + c_j)$. Using this fact, a single quadric E_Q can easily be defined for the set of planes of a given vertex [Gar99] as sum over quadrics for each face:

$$E_Q(\mathbf{v}) = \sum_i D_i^2(\mathbf{v}) = \sum_i Q_i(\mathbf{v}) = Q(\mathbf{v}) \tag{2.4.8}$$

In other words, each vertex contains accumulated information about the error for the whole local neighborhood of **v**. For the pair of vertices the cost of contraction $(\mathbf{v_i}, \mathbf{v_j}) \rightarrow \bar{\mathbf{v}}$ is simply:

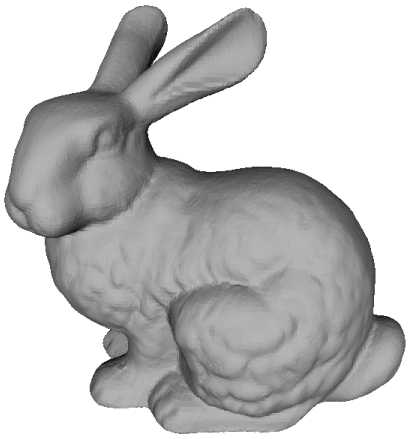
$$Q(\bar{\mathbf{v}}) = Q_i(\bar{\mathbf{v}}) + Q_j(\bar{\mathbf{v}}) \tag{2.4.9}$$

The value of $Q(\bar{\mathbf{v}})$ is a key in the min-heap. Tables 2.3 and 2.4 show an example of simplification using quadric metric. Due to complexity of the original Stanford Bunny mesh which has 69451 faces, it was first simplified to 3642 faces and then used as a reference model.

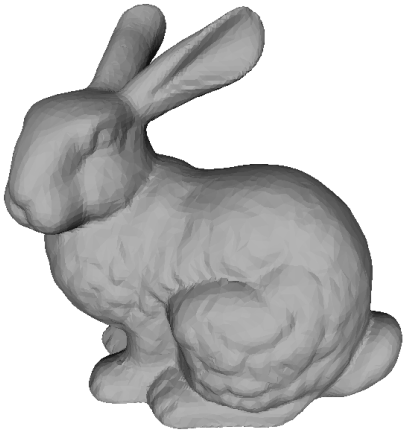
Number of faces	Size in % of the original mesh
3642 faces	94.7%
2228 faces	96.8%
1842 faces	97.3%
1152 faces	98.3%
665 faces	99.0%
130 faces	99.8%

Table 2.1: Percentage of simplification of the original Stanford Bunny with 69351 faces.

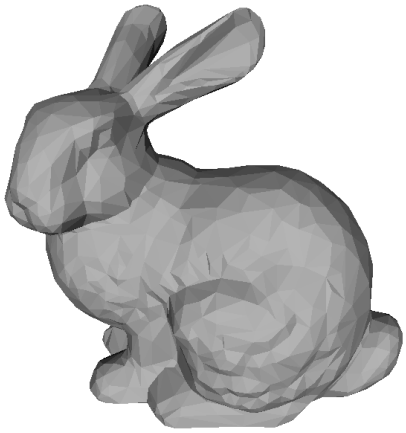
Table 2.2 shows the quality of progressive simplification. 70% of reduction is hard to distinguish from the original mesh, even the 95% is still a good approximation and features of the bunny are preserved fairly well.



(a) Faces 69451 (100%)

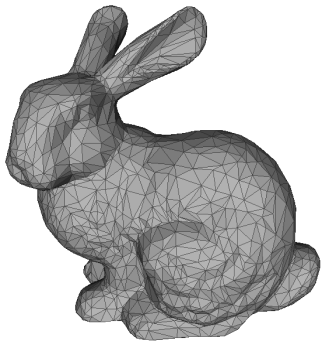


(b) Faces 18892 (30%)

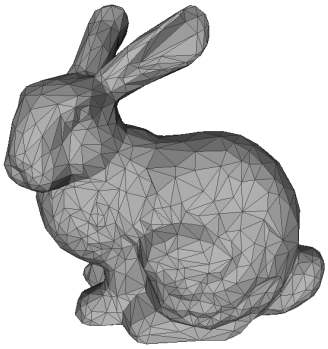


(c) Faces 3642 (5%)

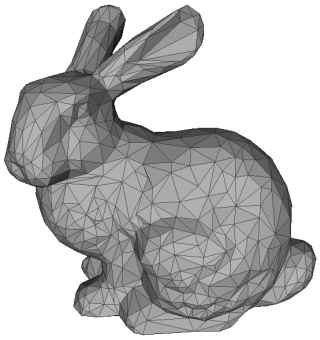
Table 2.2: Quality of the simplification of the original mesh.



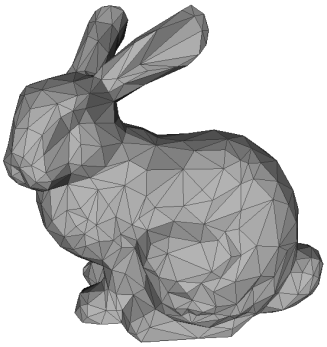
(a) Faces 3642



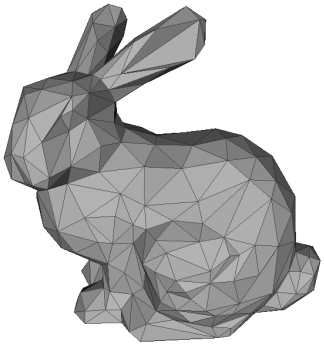
(b) Faces 2228



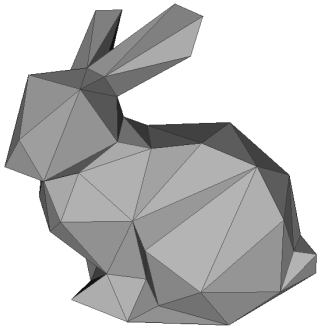
(c) Faces 1842



(d) Faces 1152

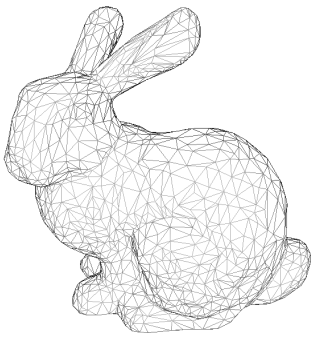


(e) Faces 655

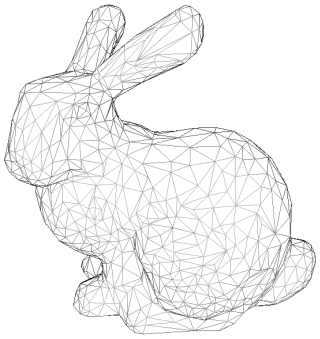


(f) Faces 130

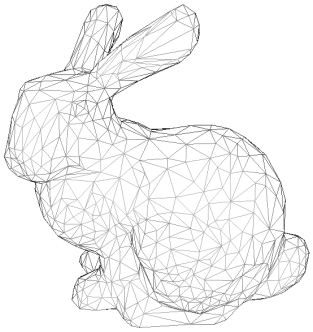
Table 2.3: Several approximations of Stanford Bunny constructed with the geometry quadric error metric.



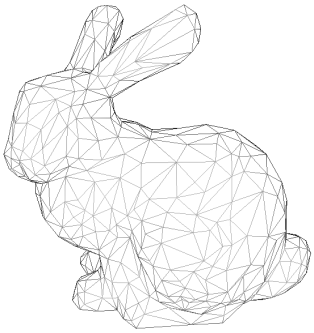
(a) Faces 3642



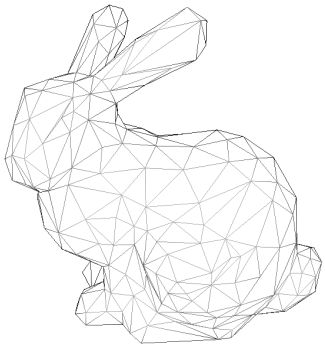
(b) Faces 2228



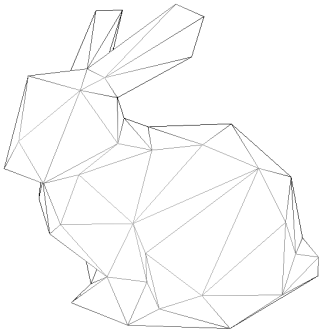
(c) Faces 1842



(d) Faces 1152



(e) Faces 655



(f) Faces 130

Table 2.4: Wireframe versions of models in Table 2.3

2.5 Vertex Placement

To perform the contraction of an edge $(\mathbf{v_i}, \mathbf{v_j}) \rightarrow \bar{\mathbf{v}}$ the calculations of a new position $\bar{\mathbf{v}}$, which is called the target, has to be done. The optimal placement strategy, therefore, is to find a point for which $Q(\bar{\mathbf{v}})$ is minimal. Fortunately, since, $Q(\bar{\mathbf{v}})$ is quadratic we are guaranteed to find an unique minimizer which is a global minimum.

$$Q(\mathbf{v}) = \mathbf{v}^T\mathbf{A}\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c \tag{2.5.1}$$

$$\nabla Q(\mathbf{v}) = 2\mathbf{A}\mathbf{v} + 2\mathbf{b} \tag{2.5.2}$$

Solving for $\nabla Q(\mathbf{v}) = 0$, the optimal position is defined:

$$\bar{\mathbf{v}} = -\mathbf{A}^{-1}\mathbf{b} \tag{2.5.3}$$

and the error:

$$Q(\bar{\mathbf{v}}) = \mathbf{b}^T\bar{\mathbf{v}} + c = -\mathbf{b}^T\mathbf{A}^{-1}\mathbf{b} + c \tag{2.5.4}$$

The function used to calculate the optimal position $\bar{\mathbf{v}}$ is the following:

```
1 virtual bool optimize(Eigen::VectorXd &result) {
2         Eigen::FullPivLU<Eigen::MatrixXd> lu = A.fullPivLu();
3         if (!lu.isInvertible())
4                 return false;
5         result = -lu.solve(b);
6         return true;
7 };
```

Listing 2.1: LU decomposition for solving a linear system.

Eigen::FullPivLU is LU decomposition of a matrix with complete pivoting, and related features. This decomposition provides the generic approach to solving systems of linear equations, computing the rank, invertibility, inverse, kernel, and determinant [Eig19].

The standard routine is based on 3 basics checks [Gar99]:

1. Try to compute $\bar{\mathbf{v}}$ (2.5.3)
2. If $\mathbf{A}$ is singular, find the optimal position along the line segment $(\mathbf{v_i}, \mathbf{v_j})$.
3. If this is not unique, select the better of $\mathbf{v_i}$ and $\mathbf{v_j}$.

In practice it is very rare that a matrix determinant is zero. Due to the limits of floating point precision. However, the function $isInvertible()$ determines which pivots should be considerd nonzero, based on a certain threshold. A matrix with determinant close to zero will be considered as not a full-rank matrix [Str88]. Consequently, step 2 or 3 has to be performed to find the optimal point.

The optimal vertex placement will tend to create closely fitting approximations of the original mesh. Therefore, the resulting meshes are shaped in a way that triangles are more equilateral and their areas are more uniform. This method is the best choice for generating fixed approximations of an original [Gar99].

Summarizing, the general placement strategy for the pair of contraction $(\mathbf{v_i}, \mathbf{v_j})$ is to always move $\mathbf{v_i}$ to $\bar{\mathbf{v}}$ and then $\mathbf{v_j}$ to the position of $\mathbf{v_i}$. It eliminates a problem of storing delta of the new vertex position.

2.6 Constraints

Simplification requires several different constrains and checks to not introduce an error in the new vertex placement position. The quality of approximation is critical for producing simplified meshes. For instance, the borders of a mesh have to be treated as a special case.

There are a few options how we can solve the problem of borders. This implementation uses a version with adding the quadric error of a plane perpendicular to a given border edge. The perpendicular plane itself defines a boundary constraint. The other option is to not touch border vertices. However, this option is problematic if a certain level of simplification has to be achieved. The number of border vertices can be even 15% of the mesh. It means that this 15% has to be transfer from the complex shapes which we would like to preserve to the simplification pool. Therefore, the quality of the mesh is sacrificed for the sake of preserving edges.

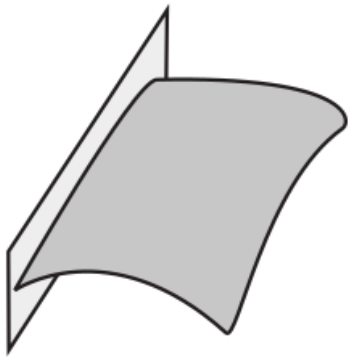


Figure 2.3: An example of the border constrain [Gar99].

The perpendicular plane constraint is very convenient in many aspects. For instance, it requires only to calculate the quadric error and add this error to the initial quadric error of each vertex on the boundary. It can be easily done in the initialization step when the heap is built. After each main iteration, the vertices are flagged if they lay on the border. Once the perpendicular error planes are accumulated in the vertices the algorithm continues the regular routine.

The perpendicular plane constraint can be additionally weight by an arbitrary constant factor. This implementation do not use the factor multiplication because of the adaptive thresholding, which ignores all edges above the current threshold level. In this version of the algorithm, the edges are consumed as soon as possible to reduce the simplification pool.

The other very important problem is vertex folding. A vertex placement can introduce a new position for a vertex which folds the neighborhood and produces degeneracies into the mesh.

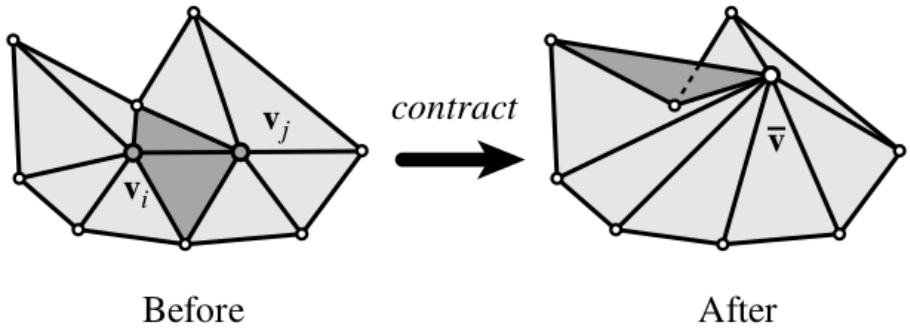


Figure 2.4: An edge contraction which causes the mesh to fold over on itself [Gar99].

Before performing a contraction, the algorithm has to check, if the new position is a valid one. For example, Figure 2.4 shows the degenerated new position of $\bar{\mathbf{v}}$ which folds one of the faces into the darkened area. In this case, the contraction is stopped for this particular edge. To detect those kind of situations, the normals of faces around $\mathbf{v_i}, \mathbf{v_j}$ are examined. If the face normal changes by some threshold level, the contraction is assumed to introduced flipping and is discarded.

To determine if a flipping occurred, the two checks are preformed. First, checking if the area of newly created face is sufficient. Meaning, investigate the angle between edges. If it is bigger than some threshold, edges are almost co-linear. If a triangle is defined as $T = (\mathbf{p}, \mathbf{q}, \mathbf{r})$ then:

$$\mathbf{u} = \frac{\mathbf{r} - \bar{\mathbf{v}}}{\|\mathbf{r} - \bar{\mathbf{v}}\|} \tag{2.6.1}$$

$$\mathbf{v} = \frac{\mathbf{q} - \bar{\mathbf{v}}}{\|\mathbf{q} - \bar{\mathbf{v}}\|} \tag{2.6.2}$$

$$t = |\mathbf{u} \cdot \mathbf{v}| \tag{2.6.3}$$

where $\mathbf{u}$ and $\mathbf{v}$ are unit vectors and $\bar{\mathbf{v}}$ is the new optimal position. Variable t caries the notion of angle between two vectors, if t is bigger than 0.999 the flipping was introduced and two new edges are very close to be co-linear.

The second check investigates normals. Using previously defined vectors $\mathbf{u}$ and $\mathbf{v}$ and triangle T with its normal as $\mathbf{n}_T$ the definition is as follows:

$$\mathbf{n} = \frac{\mathbf{u} \times \mathbf{v}}{\|\mathbf{u} \times \mathbf{v}\|} \tag{2.6.4}$$

$$c = \mathbf{n} \cdot \mathbf{n}_T \tag{2.6.5}$$

If c is smaller than 0.2 the contraction is rejected. Basically, it means that the new face is going to flip and is close to be perpendicular to the neighboring face. In many cases it introduces inconsistencies. In this situation the removal is also rejected.

To correctly investigate flipping, the whole silhouette of neighbors of a given new vertex position has to be checked. The test check has to pass for all of them to accept a decimation.

2.7 Summary of Garland's Algorithm

Below, complete algorithm in 5 steps is presented. The next chapter elaborates how this concept is incorporated in multi-threaded approach.

Algorithm [Gar99]:

1. Select a set of candidate vertex pairs $(\mathbf{v_i}, \mathbf{v_j})$.
2. Allocate a quadric Q_i for each vertex $\mathbf{v_i}$.
3. For each face compute a quadric Q_i. Add this fundamental quadric to the vertex quadrics Q_i, Q_k, Q_l and optionally weight it appropriately.
4. For each candidate pair $(\mathbf{v_i}, \mathbf{v_j})$:
   (a) Compute $Q = Q_i + Q_j$.
   (b) Select a target position $\bar{\mathbf{v}}$.
   (c) Apply consistency checks and penalties.
   (d) Place pair in heap keys on cost $Q(\bar{\mathbf{v}})$
5. Repeat until the desired approximation is reached:
   (a) Remove the pair $(\mathbf{v_i}, \mathbf{v_j})$ of least cost from the heap.
   (b) Preform contraction $(\mathbf{v_i}, \mathbf{v_j}) \rightarrow \bar{\mathbf{v}}$
   (c) Set $Q_i = Q_i + Q_j$.
   (d) For each remaining pair $(\mathbf{v_i}, \mathbf{v_j})$, compute target position and cost as in step 4; update heap.

The algorithm above is the main core of the implementation. Every thread will perform the simplification using quadric error metrics, based on Garland's work. In the next chapter, a brief introduction to the extended version of the algorithm which includes color and normals is given.

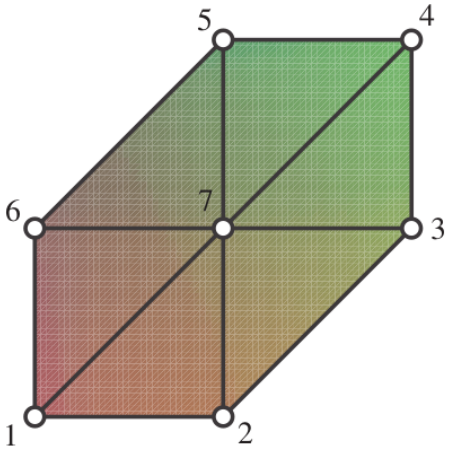
Chapter 3

Extended Simplification Algorithm

The previous implementation includes only geometric error. In this chapter more attributes to improve simplification are incorporated. Using additionally color and normals, the decimation of planar surfaces can be easier achieved and gives better results. However, in the noisy environment of color (which is usually the case in scans), gradient guided simplification may produce poor results.

3.1 Design

In this section, the main assumption is that each vertex is additionally associated with color $\mathbf{c} = [r\ g\ b]^T$ and normal $\mathbf{n} = [nx\ ny\ nz]^T$. For the sake of simplicity, lets consider an example with geometry and color only. Assume that each vertex is attributed with $\mathbf{v} = [x\ y\ z\ r\ g\ b]^T$. Figure 3.1 shows an example of a mesh like that. A triangle is defined as $T = (\mathbf{p}, \mathbf{q}, \mathbf{r})$ with edges $\mathbf{h} = \mathbf{q} - \mathbf{p}$ and $\mathbf{k} = \mathbf{r} - \mathbf{p}$. Now, because the problem is more that 3 dimensional, a different method to calucalte orthogonal vectors to the plane defined by the face is used. Simply calculating a corss product of two vectors will not work. Gram-Schmidt orthogonalization method [Str88] is used to solve this problem.



(a) Simple colored mesh

Vertex	Position			RGB Color		
1	[0	0	0	0.7	0.3	0.3]
2	[1	0	0	0.7	0.4	0.3]
3	[2	1	0	0.5	0.5	0.3]
4	[2	2	0	0.3	0.5	0.3]
5	[1	2	0	0.3	0.4	0.3]
6	[0	1	0	0.5	0.3	0.3]
7	[1	1	0	0.5	0.4	0.3]

(b) Table of vertices

Figure 3.1: A triangulated hexagon with color values at each vertex [Gar99].

Therefore, two orthogonal to each other vectors $\mathbf{e_1}, \mathbf{e_2}$ are defined as:

$$\mathbf{e_1} = \mathbf{h}/\|\mathbf{e_1}\| \tag{3.1.1}$$

$$\mathbf{e_2} = \frac{\mathbf{k} - (\mathbf{e_1} \cdot \mathbf{k})\mathbf{e_1}}{\|\mathbf{k} - (\mathbf{e_1} \cdot \mathbf{k})\mathbf{e_1}\|} \tag{3.1.2}$$

$\mathbf{e_1}, \mathbf{e_2}$ are unit-length vectors which form a local coordinate system with $\mathbf{p}$ as the origin. Those vectors describe a plane like object in R^6.

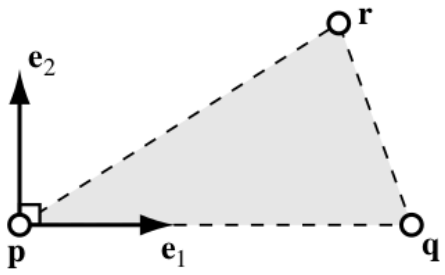


Figure 3.2: Orthonomal vectors [Gar99].

Now, the distance from an arbitrary point $\mathbf{v} \in \mathbf{R}^n$ to the plane created by the face T will be presented. The squared distance of the vector $\mathbf{u} = \mathbf{p} - \mathbf{v}$ is defined as [Gar99]

$$\|\mathbf{u}\|^2 = \mathbf{u}^T\mathbf{u} = (\mathbf{u}^T\mathbf{e_1})^2 + ... + (\mathbf{u}^T\mathbf{e_n})^2 \tag{3.1.3}$$

Rearranging the equation gives:

$$(\mathbf{u}^T\mathbf{e_3})^2 + ... + (\mathbf{u}^T\mathbf{e_n})^2 = \|\mathbf{u}\|^2 - (\mathbf{u}^T\mathbf{e_1})^2 - (\mathbf{u}^T\mathbf{e_2})^2 \tag{3.1.4}$$

The left hand side is the squared distance of $\mathbf{u}$ along all the axes perpendicular to the plane of T. This is the distance between this vector $\mathbf{v}$ and the plane T.

$$D^2 = \mathbf{u}^T\mathbf{u} - (\mathbf{u}^T\mathbf{e_1})(\mathbf{u}^T\mathbf{e_1}) - (\mathbf{u}^T\mathbf{e_2})(\mathbf{u}^T\mathbf{e_2}) \tag{3.1.5}$$

The quadric metric is defined as $Q(\mathbf{v}) = \mathbf{v}^T\mathbf{A}\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c$ where:

$$\mathbf{A} = \mathbf{I} - \mathbf{e_1}\mathbf{e_1}^T - \mathbf{e_2}\mathbf{e_2}^T \tag{3.1.6}$$
$$\mathbf{b} = (\mathbf{p} \cdot \mathbf{e_1})\mathbf{e_1} + (\mathbf{p} \cdot \mathbf{e_2})\mathbf{e_2} - \mathbf{p} \tag{3.1.7}$$
$$\mathbf{c} = \mathbf{p} \cdot \mathbf{p} - (\mathbf{p} \cdot \mathbf{e_1})^2 - (\mathbf{p} \cdot \mathbf{e_2})^2 \tag{3.1.8}$$

The matrix $\mathbf{A}$ from 3.1 is then:

$$\begin{bmatrix} 0.06 & 0 & 0 & 0 & -0.59 & 0 \\ 0 & 0.23 & 0 & 1.15 & 0 & 0 \\ 0 & 0 & 6.00 & 0 & 0 & 0 \\ 0 & 1.15 & 0 & 5.77 & 0 & 0 \\ -0.59 & 0 & 0 & 0 & 5.94 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6.00 \end{bmatrix} \tag{3.1.9}$$

Building the quadric and finding the optimal position is exactly the same like in the geometry case. To use a different metric, a proper orthonormal vectors in $\mathbf{R}^n$ have to be provided.

Surface type	Vertex type	dim $\mathbf{A}$	No. unique coeff.
Geometry only	$[x\ y\ z]^\mathsf{T}$	3×3	$\binom{5}{2} = 10$
Geometry & 2-D texture	$[x\ y\ z\ s\ t]^\mathsf{T}$	5×5	$\binom{7}{2} = 21$
Geometry & Gouraud color	$[x\ y\ z\ r\ g\ b]^\mathsf{T}$	6×6	$\binom{8}{2} = 28$
Geometry, color & normal	$[x\ y\ z\ r\ g\ b\ u\ v\ w]^\mathsf{T}$	9×9	$\binom{11}{2} = 55$

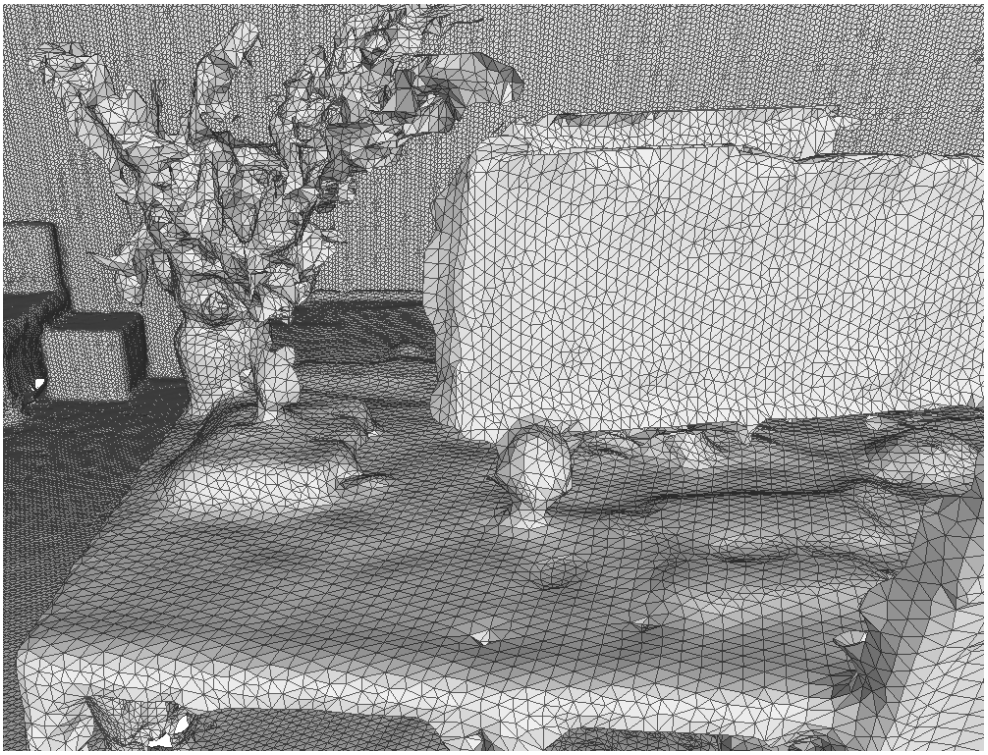
Figure 3.3: Summary of common extended quadric types [Gar99].

3.2 Results

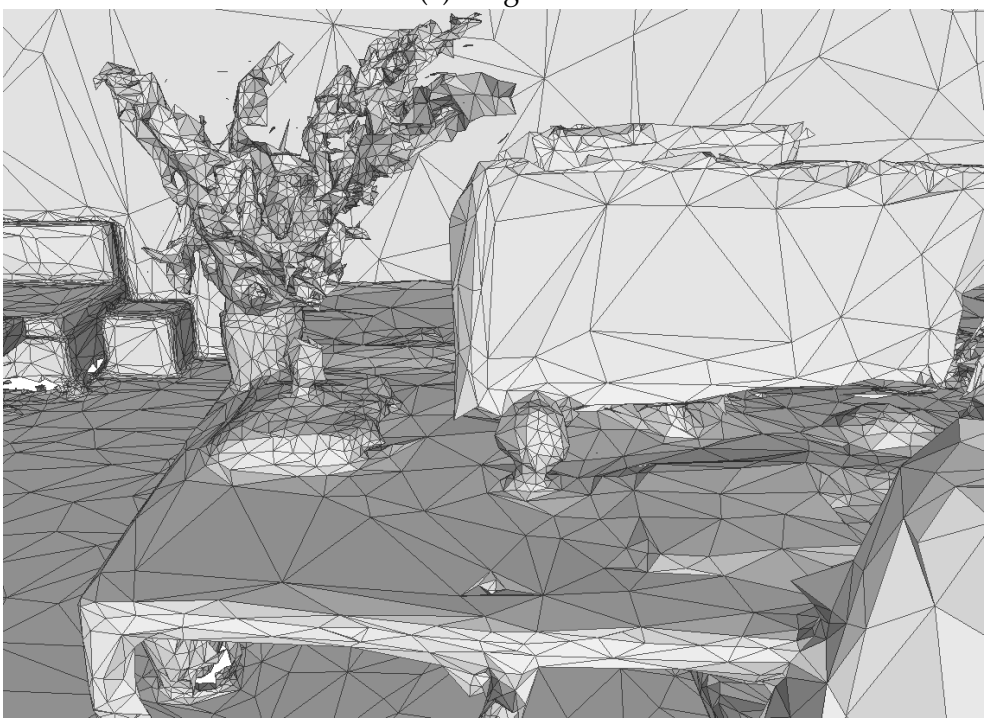
Table 3.2 shows the results of using color and geometry metric error. It can be easily noticed that the simplification follows the color pattern. In some cases it is a desired features, however, the gradient can introduce noise.

In the case of $3D$ scanning gives vertices with several attributes; for instance color, normal and geometry. Unfortunately, the color introduces quite substantial error. If the color gradient was constant in most of the places, the whole simplification would hugely benefit from it. However, not only a camera introduces an error, but also a SLAM algorithm. Therefore, either normals, color, geometry are incorporated all together or used separately.

As it can be seen in the Table 3.1, complex shapes like plants or a small fan in front of the monitor preserve their initial complexity. However, flat surfaces like the desk or the monitor are significantly simplified. Therefore, the amount of removable faces, to achieve a particular simplification level, was transfered from the complex shapes pool to the flat surface pool. Complex shapes will be simplified only in the case when planar surfaces are as simply as possible and removal is not longer available.



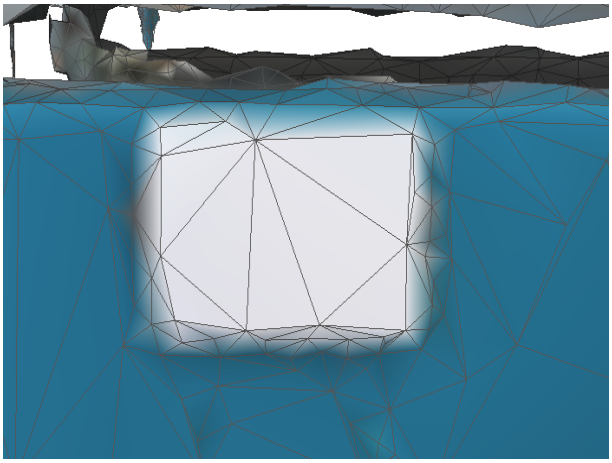
(a) Original



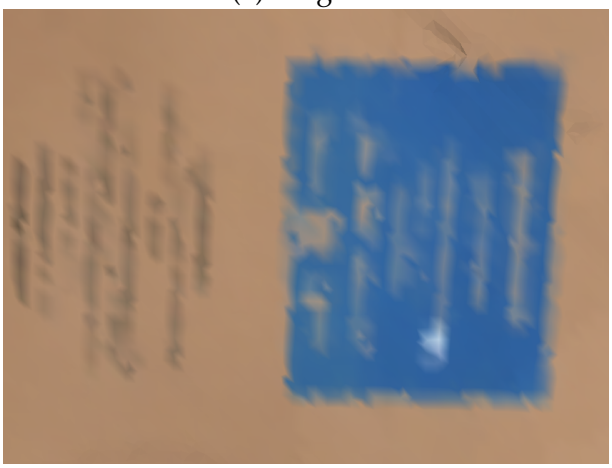
(b) Color, geometry and normal simplification

Table 3.1: Comparison of simplification with all attributes [geometry, color, normal] with 87% of reduction to the original mesh.

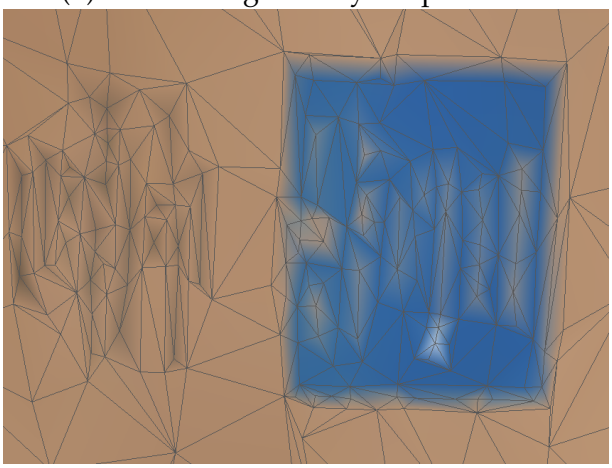
(a) Original



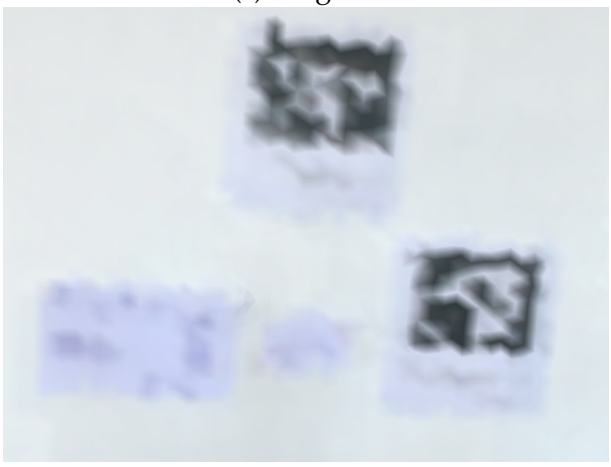
(b) Color and geometry simplification



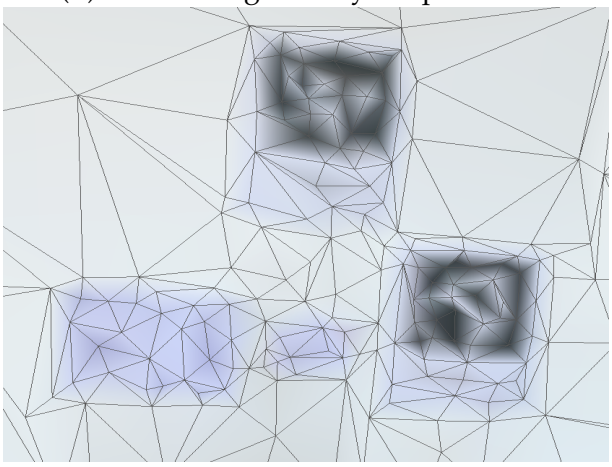
(c) Original



(d) Color and geometry simplification



(e) Original



(f) Color and geometry simplification

Table 3.2: Comparison of gradient guided simplification [geometry, color].

Chapter 4

Parallel Simplification Algorithm

In this chapter, the approach to design a parallel version of Garland's simplification algorithm is introduced. First, the Producer-Consumer design pattern with libraries and the implementation ideas making it thread-safe are described. Next, the analysis of the speedup and potential problems which can arose during an execution are shown. Finally, summary of the approach and comparison to different algorithms is given.

4.1 Producer Consumer Pattern

The producer-consumer pattern is an optimal way to separate workload of processing data from the work to produce them. In the other words, it divides the problem into two major components, connected usually by a queue. This is a classic example of a multi-process synchronization problem. The process separation gives clear view at the problem and tasks. A producer is placing items in a queue, and a consumer removes each task from the queue and processes the data. This decoupling means that two components are completely independent [Gra02]. In the case when the queue is full, locking procedure is used to wait for consumers to process tasks.

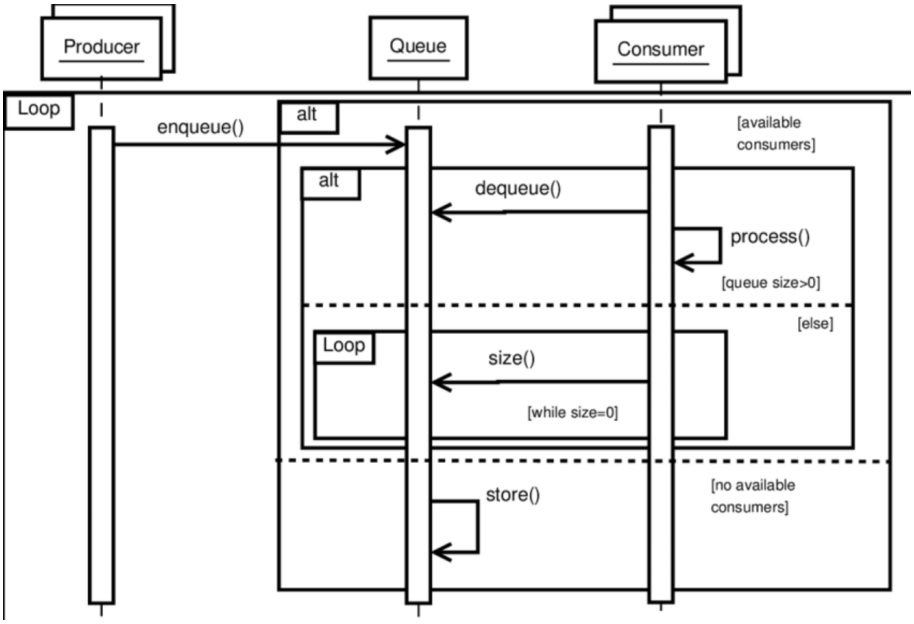


Figure 4.1: UML sequence diagram for the procuder-consumer pattern implementation [RT17].

Figure 4.1 shows how such a process could look like. There are shown three main components; producer, synchronization object (the queue) and consumer. In the case when the buffer is either full or empty, waiting spin[1] is used to hold execution till the moment when more tasks are available for a consumer.

In this implementation a generalized version of the pattern is used, which assumes multiple producers and consumers operating on a single fixed buffer. The main idea is to create a separate tasks, which later are consumed by active threads from the thread-pool[2]. To create those tasks, clustering of the mesh is introduced.

[1] A procedure to constantly check if the queue has new tasks to process. This is a common technique used in parallel programming.

[2] A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work, in this case simplification tasks from the queue.

4.2 Producer Design

To cluster a mesh into sub-boxes, a simple method of dividing the global bounding box is used. When the mesh is read, the global bounding box is calculated, which later is divided into predefined number of clusters.

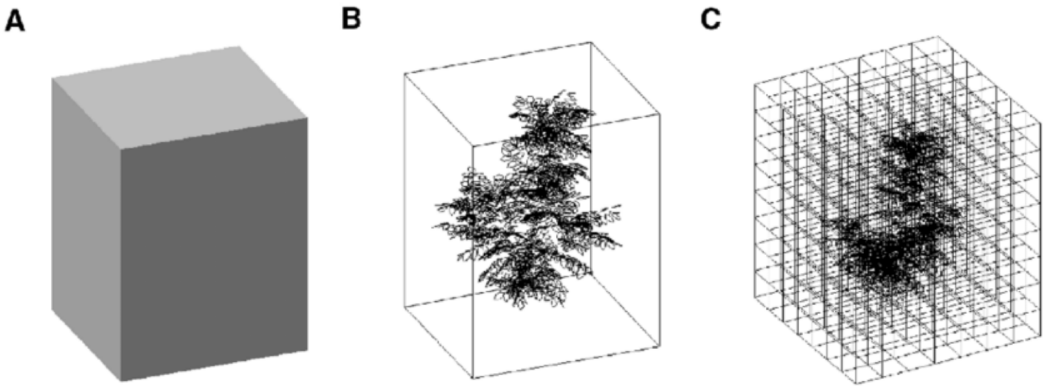


Figure 4.2: Example of clustring a mesh into 7x7x7 clusters.

Figure 4.2 shows an example with arbitrary $7x7x7 = 343$ clusters, where each dimension of the main bounding box $[B]$ was divided into 7 equal parts $[C]$. When clustered boxes are calculated, the membership check to which cluster belongs a face is performed. In the case when a face intersects multiple clusters the voting procedure is done. Which is defined as follows; if two vertices of a face belong to the same cluster this box is selected as a holder of the face. If each vertex belongs to a different cluster then one box is randomly selected. The pseudocode of the producer is the following:

```
1  void function produce {
2      clusters = getClusters(size, masterMesh);
3      for (cluster : clusters){
4          set faceCluster to empty
5          for (face : cluster.elements){
6              if vote(face) is true
7                  add face to faceCluster
8          }
9          add faceCluster to queue
10     }
11 };
```

Listing 4.1: C style psuedocode of a producer.

In the line 7 the $vote()$ function is used to assign the correct cluster *id* and membership of a face. The line 9 inserts the $faceCluster$ to the queue for further processing.

4.3 Consumer Design

The consumer process is desing in a way to utilize funcionality of `boost::thread_group` and `boost::asio::io_service`. The thread group is a class which helps with thread management and provides a container for easy grouping of threads to simplify several common thread creation and management idioms [WE19]. A user has to specify how many threads will be created and added to the thread-pool. Those threads operate directly on the queue, removing tasks and processing data. In this case they are preforming simplification of a mesh on each cluster independently.

```
1 void function task {
2     garland = QSlim();
3     garland->setClustersAABBs();
4     garland->initialize<QuadricError>();
5     garland->simplify<QuadricError>();
6 };
```

Listing 4.2: C style psuedocode of a task for a consumer.

Each task gets a single cluster to process. The *QuadricError* is a type of a quadric error metric which specifies attributes for the calculations like; geometry, color, normals. The class *QSlim* is responsible for all simplification operations and was elaborated in the previous sections.

All decimating operations executed by threads are independent, except those on the borders. If one vertex of a face belongs to a different cluster and this cluster is processed by a different thread, a clever locking strategy has to be used. The main idea is to lock[3] the whole neighborhood of an edge. To avoid deadlocks, a specialized version of `boost::recursive_mutex` is used. It allows to be locked multiple times by the same thread. In the border situation, the best solution for obtaining a lock is to use the method `bool try_lock()` which returns immediately false if the mutex is already locked by a different thread. It means that the given neighborhood is already processing by a different thread and the simplification process for the current thread is terminated.

`boost::asio::io_service` is responsible for the whole producer consumer work-flow and is the centerpiece in this implementation. The service nicely utilizes functionality of a thread-pool. The method *post*() creates tasks and pushes them to the queue where later they are processed by threads from the thread-pool.

[3]Locking is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution.

4.4 Design

The algorithm is based on Garland's simplification quadric error metric algorithm. The main difference is clustering and parallel consuming parts of a mesh with adaptive thresholding for each iteration. This approach guarantees global update and accumulation of quadrics for each iteration in such a way that planar surfaces are always decimated first. The main goal of this work, is to transfer from global pool of vertices for a mesh to two separate pools, one for complex shapes and one for planar surfaces. A vertex pools are just conceptual terms which mean the number of vertices which we want to remove to achieve our convergence criteria. However, to keep complex shapes intact, vertices from planar surfaces have to be removed as much as possible to maintain a given trade-off in pools. Therefore, outer loop of the algorithm, which increases the threshold level, is so crucial. The reconstruction always creates a mesh with evenly distributed triangles. This assumption, drives the design of the algorithm. Complex shapes have high quardic error. Appropriately manipulated threshold can achieve the desired transfer goal.

To improve time of the simplification convergence, aggressiveness can be always increased. The only problem is the final quality of the approximation. The iterative nature of using Garland's algorithm and adaptive thresholding, which targets only planar surface, does not always hold in this case. However, it allows to save around 50% time of processing by slightly increasing aggressiveness. It can be easily modify be program parameters.

4.5 Resutls

In this section, results of multi-threaded experiments of the algorithm are elaborated. Additionally, the time investigation of each execution in multiple scenarios is shown. The tests were performed on a machine with 32 GB of RAM and Intel i7-6700 CPU 3.40GHz with 8 physical cores on a mesh with 982624 faces and 517715 vertices. Four different setups were ran with the objective to achieve reduction level 85% of the original mesh:

Number of clusters	Number of threads in the pool	Total time processing
1	1	151.611s
8	8	75.557s
27	27	57.405s
27	8	58.957s

Table 4.1: 4 different test setups

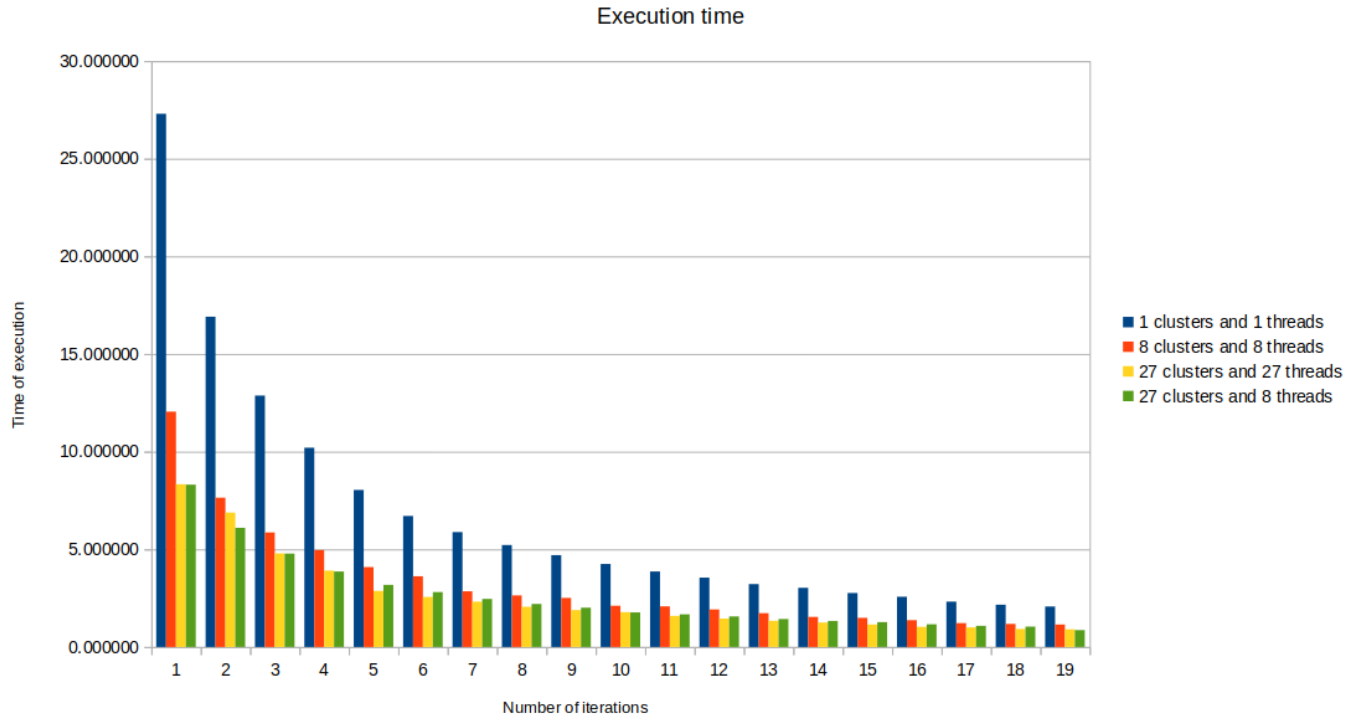


Figure 4.3: Time of execution in seconds with different number of threads and clusters.

The Figure 4.3 shows how much time does it take to simplify a mesh in one iteration. In total, there were 19 iterations to achieve the 85% of simplification. The one iteration in this case, is the full pass of the simplification algorithm. It means; to wait for all active threads to finish processing sub-meshes. Naturally, the time is getting smaller

with each iteration because of less vertices to process. As it is expected, the single threaded version performs very poorly. Using more than 1 thread can achieve better results with the best speedup around 3.5.

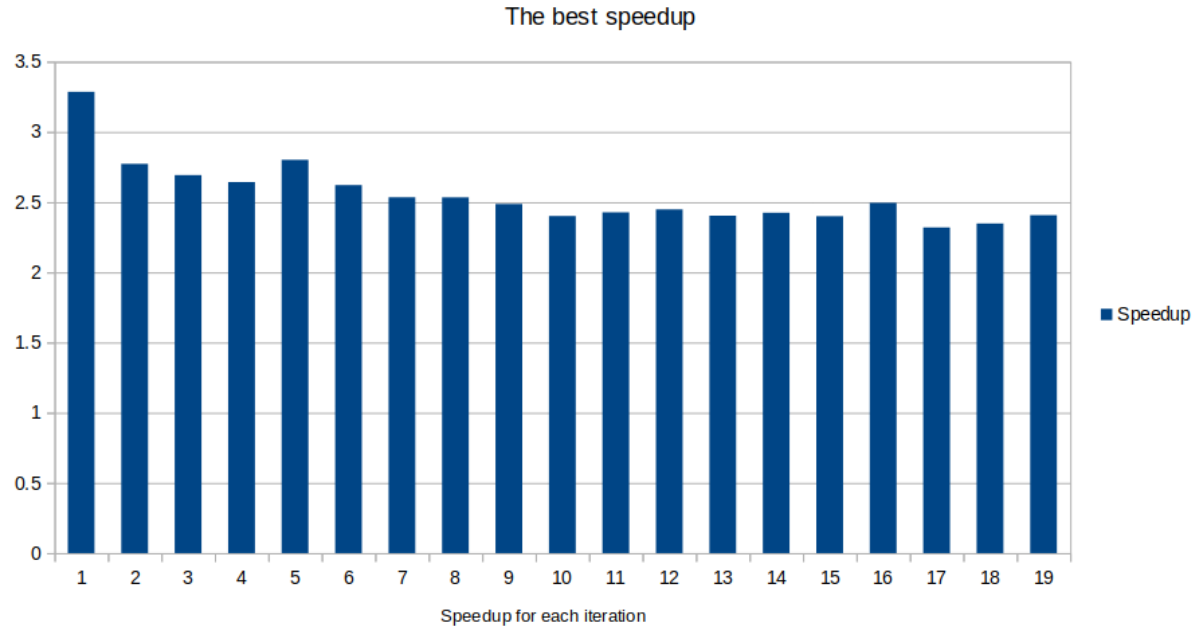


Figure 4.4: The best speedup for each iteration

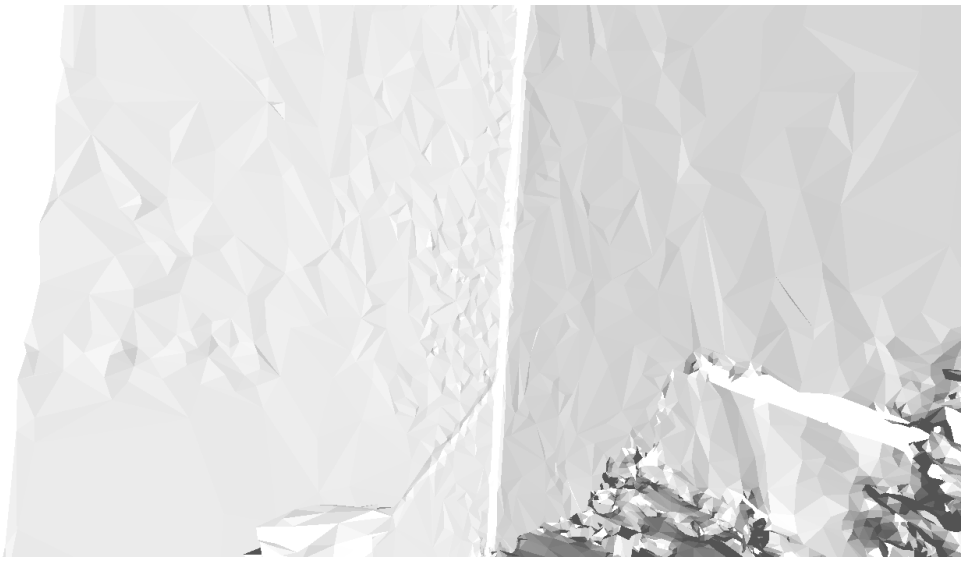
The Figure 4.4 desciribes the best speedup for each iteration. The value oscillates around 2.5 and 3 which is a decent result for this complex algorithm. Moreover, in the Figure 4.3 the Amdahl's law is visible in practice. Incerasing number of threads does not improve speed up [Amd67].

To avoid synchronization overhead deep coping of a sub-meshes was tried. Those deeply copied sub-meshes were passed to processing threads. The problem with this solution is merging sub-meshes into the master mesh after simplification. To preserve local geometry on the edges the process of decimation need not to change or remove them. It gives a rise to another problem with convergence. Moreover, merging meshes takes time and the algorithm does not gain much performance speedup using this strategy.

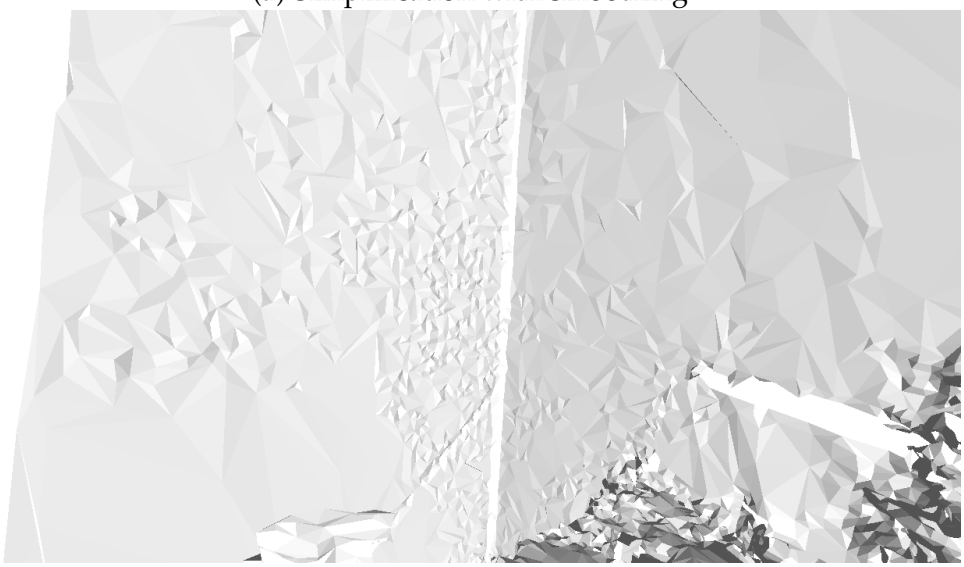
Summarizing, the parallel execution of clusters gives a desired speedup. The algorithm is able to process big meshes up to a few million of faces in reasonable time, which is necessary for streaming purposes and a production usage.

4.6 Taubin Smoothing

During the research and later implementation of the algorithm, faster convergance to selected level of reduction was noticed, if a mesh smoothing is used. For purposes of this work, Taubin Smoothing algorithm was used. The method is a linear low-pass filter that removes high curvature variations and does not produce shrinkage [Tau95].



(a) Simplification with smoothing



(b) Simplification without smoothing

Table 4.2: Comparision of the smoothing effect for 85% simplification.

Iterations	Lambda	Mu
7	0.5	-0.67

Table 4.3: Taubin algorithm parameters.

The simplification was ran on the mesh with 396277 faces and 208825 vertices. The mesh was reduced to 50009 faces and 29347 vertices. In the Table 4.4 it can be seen that the speedup compared with the single core is around 1.6. Additionally, in the Table 4.2 shows that planar surfaces look better after simplification with smoothing.

Time with smoothing	Time without smoothing
27.73 s	43.06 s

Table 4.4: Taubin algorithm parameters.

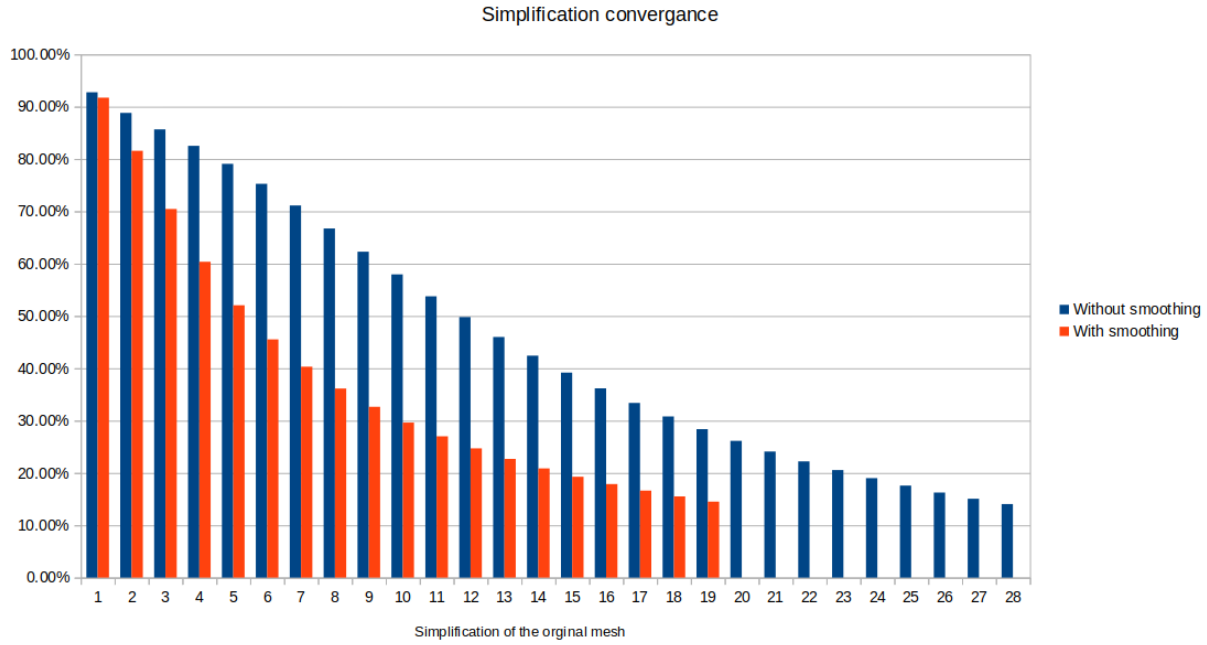


Figure 4.5: Convergance to 15% of the original mesh.

Figure 4.5 shows comparison of the algorithm execution with and without smoothing. In the case with smoothing, the convergance takes only 19 full iterations, whereas, for a regular version it takes 28. It means that with a small overhead, the complexity of Taubin's algorithm is linear $O(n)$ in the number of vertices [Tau95], a better quality of the simplified mesh and additional speedup can be obtained.

4.7 Summary of the Algorithm

1. Smooth mesh using Taubin's algorithm (optional).
2. Cluster the mesh:
   (a) Split the mesh according to number of selected clusters; for example 2x2x2=8.
   (b) Add a task for each cluster to the queue.
3. Wait for all threads from the thread-pool to finish processing simplification algorithm [QSlim] ran on all clusters. Each thread does:
   (a) Build the heap with edges.
   (b) Decimate edges till reaching the threshold level.
4. Update the mesh structure:
   (a) Remove faces flagged as invalid.
   (b) Reset properties for each remaining face and vertex.
5. Repeat from the second step till convergence.

As it is shown above, the parallel algorithm nicely wraps Garland's algorithm in the third step. Procedure from the second step is repeated till either reduction level is reached, or the number of iterations is exceeded. To achieve a nice planar simplification the adaptive threshold level is used. The third step is terminated for a given thread if the cost level is higher than the threshold.

4.8 Comparison to Commercially Available Products

Most of commercially available algorithms use only geometry to generate an approximation of a mesh. Libraries like OpenMesh do not provide an API to use all attributes of a vertex. Moreover, increasing the number of constraints does not help. The problem is not solved jointly like in the case of this work. Additionally, those algorithms simplify a mesh globally, which introduces almost even decimation for all surfaces. Below is shown a comparison between this method and several different approaches available commercially. The benchmark was to reduce the original mesh 85%.

Figure 4.6: The original mesh.

Figure 4.6 shows the original mesh with evenly distributed faces. The amount of faces used for the wall is unnecessary. Therefore, a successful simplification can be applied to this surface.

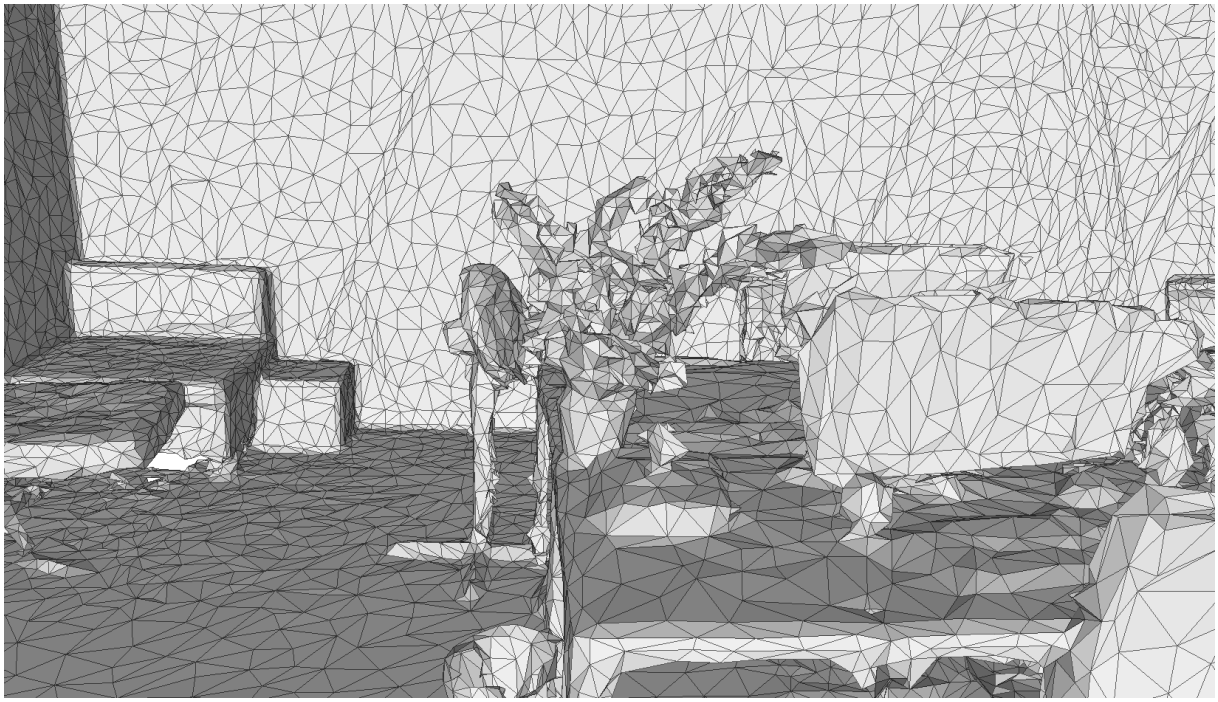


Figure 4.7: Fast-Quadric-Mesh-Simplification algorithm.

Figure 4.7 shows Fast-Quadric-Mesh-Simplification algorithm by Sven Forstmann, which is a Github implementation of memory efficient and very fast edge collapse mesh simplification method. According to the creator it is around 4 times faster than Meshlab version. Figure 4.7 shows the result where it can be seen that all surfaces are decimated evenly. Moreover, the algorithm is based only on vertex iteration with adaptive thresholding without building a heap of edges.

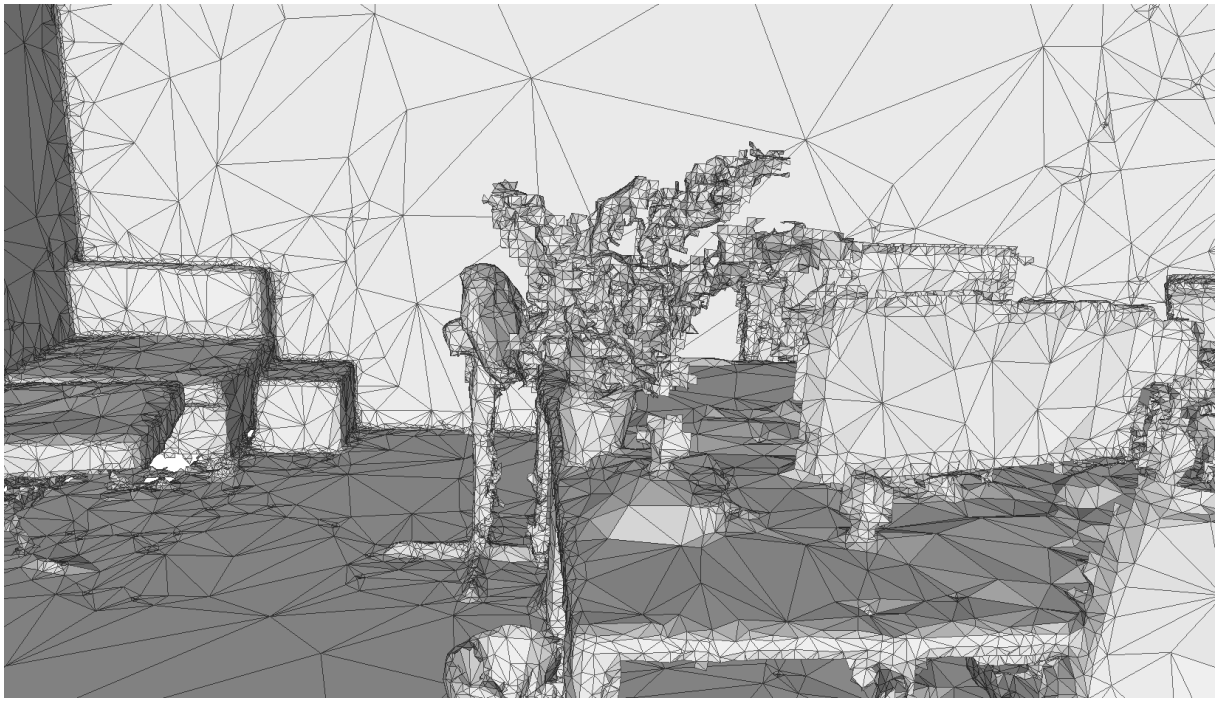


Figure 4.8: OpenMesh with geometry and normals.

Figure 4.8 shows the usage of OpenMesh library, where the decimation was done using geometry and normals. Because of that, border edges were not touched. In this case, borders can produce errors in the approximation, like spiky edges. Border preserving is mostly visible on the plant structure or the monitors edges. This method gives the poorest result, however, it is almost as fast as Fast-Quadric-Mesh-Simplification.

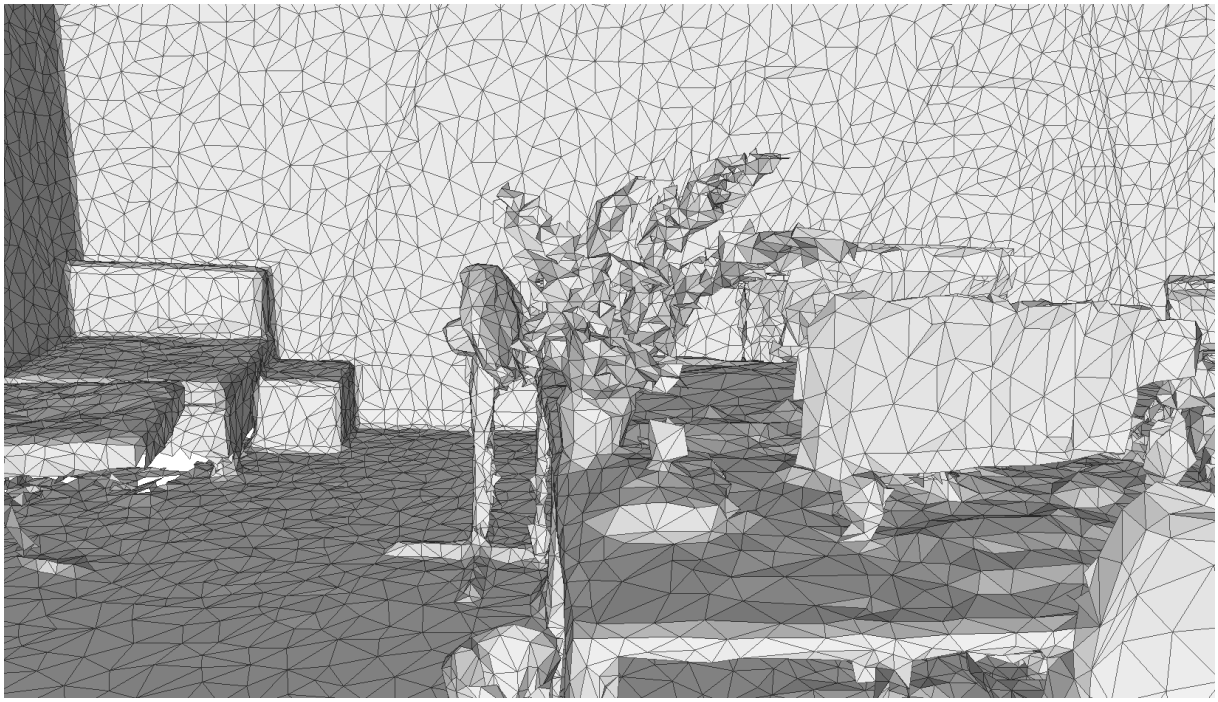


Figure 4.9: MeshLab version of simplification.

As it can be seen in the Frigure 4.9 the approximation is very similar to Figure 4.7. However, the MeshLab version is slower than Fast-Quadric-Mesh-Simplification algorithm.

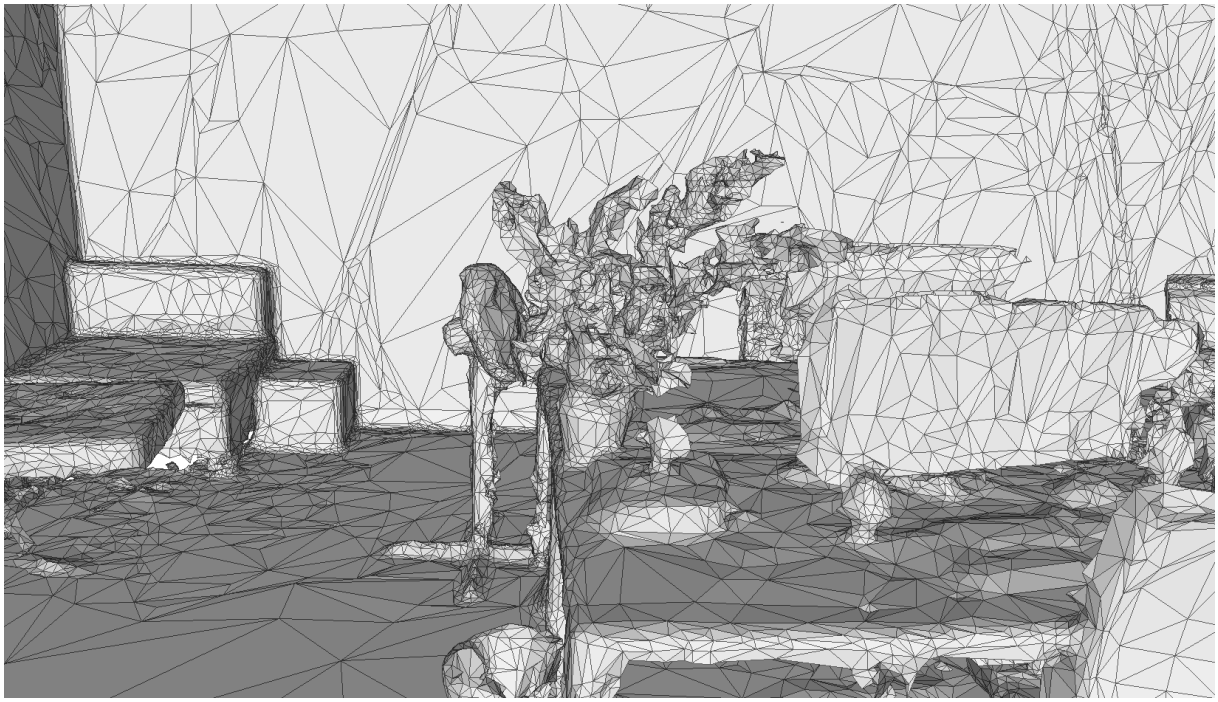


Figure 4.10: Parallel QSlim with adaptive thresholding algorithm using only geometry.

Figure 4.10 shows approximation produced by the version of the algorithm elaborated in the chapter 4. The complex shapes are preserved much better than in all previous versions. However, there is still room for even more aggressive decimation of planar surfaces. To achieve it, we need to use more attributes in our joint optimization.

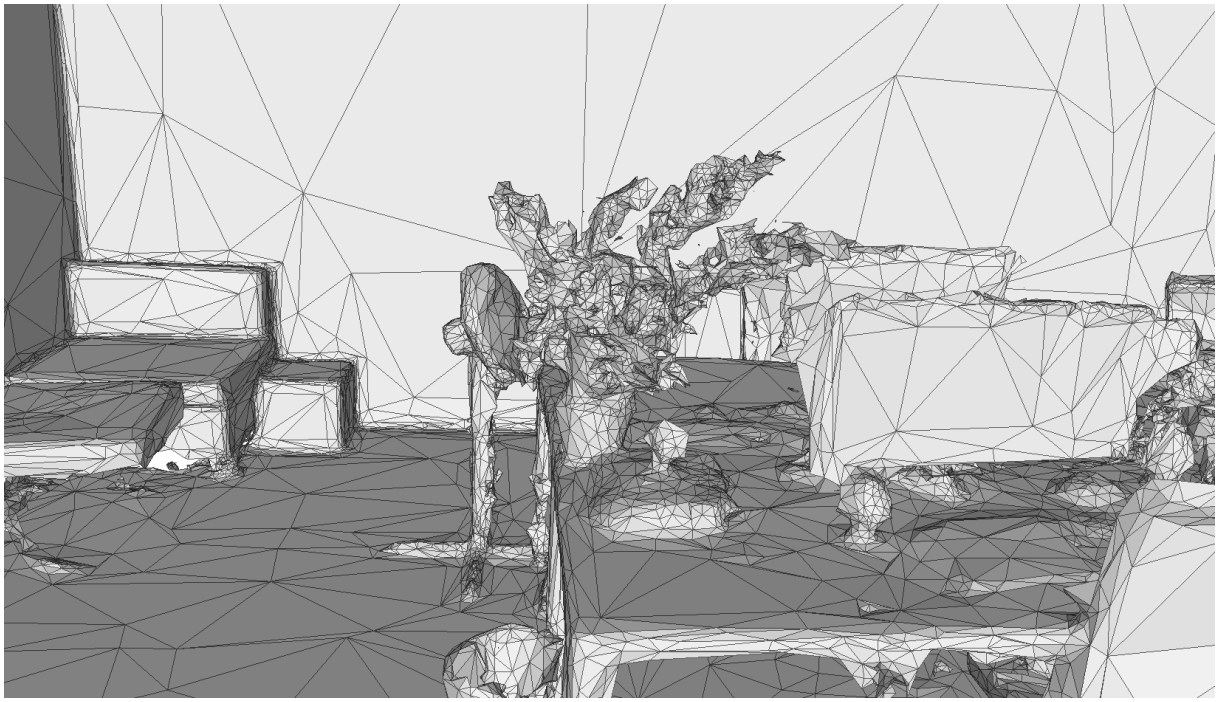


Figure 4.11: Parallel QSlim with adaptive thresholding algorithm using geometry, color and normals.

Figure 4.11 shows the best approximation under the constrain of decimating most of the planar surface. All complex shapes are almost intacted from the original mesh. The faces which describe the wall are as big as possible. Moreover, iterative nature of the algorithm, allows to level small surface elevation changes. For instance, the interior part of the pouf near the wall. In the Figure 4.10 it can be seen that the noise and level changes, meaning high quadric error, prevents this region from being simplified. In the version with all attributes $[geometry, normal, color]$, the interior part is nicely leveled.

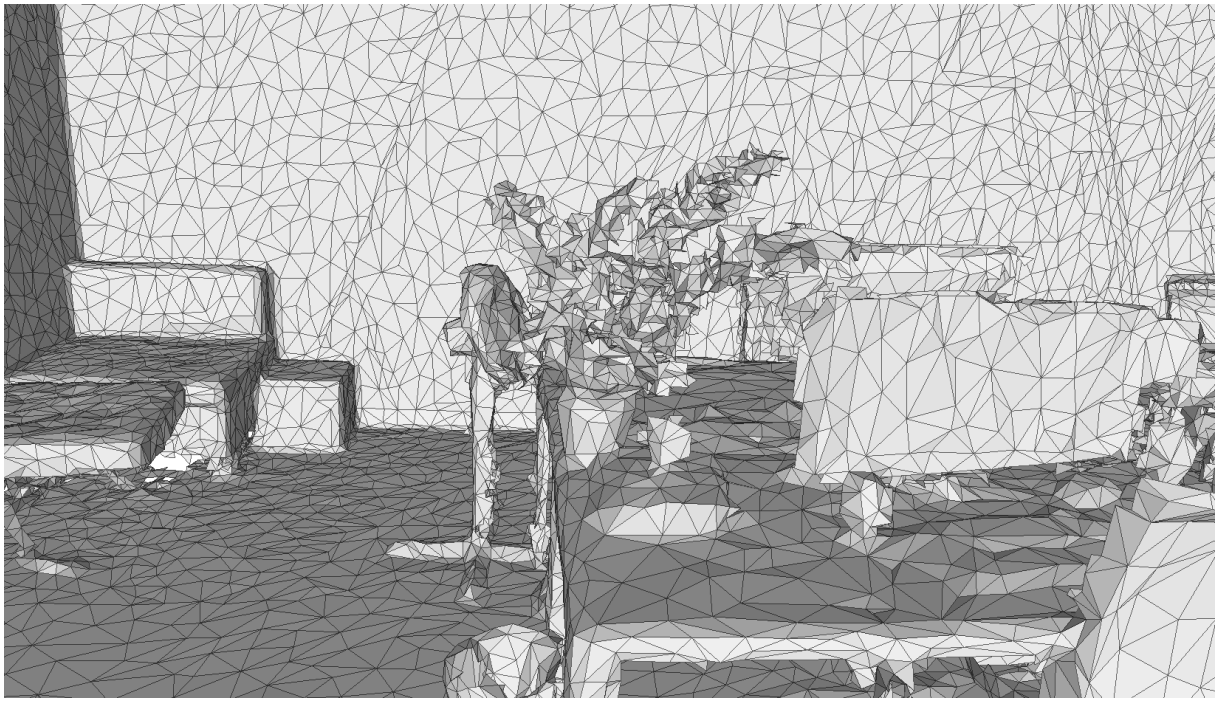


Figure 4.12: RapidCompact version of simplification.

For the completeness, a paid version of simplification algorithm was used. The algorithm was provided by the company RapidCompact. Figure 4.12 shows that the result is almost exactly the same, like in the case of the MeshLab and Fast-Quadric-Mesh-Simplification algorithm.

Chapter 5

Conclutions

The goal of this work was to create a parallel algorithm which would simplify planar surfaces and at the same time preserve high level of details in areas with complex geometry. The results show that the goal was achieved. Complex shapes like plants on desks are almost entirely preserved, whereas, walls or floors are described with a few big triangles. Appendix A shows the results where the best approximation is generated using quadric error metric with all vertex's attributes. However, at the same time, this metric is the slowest one, because of the number of parameters to jointly optimize for every edge.

Summarizing, the algorithm is able to generate high quality progressive meshes in reasonable time, which was one of the most important aspects of this work. The results are promising and in some cases are better than commercially available products. The time of processing is the biggest flaw of the approach. This problem can be reduced by increasing aggressiveness, however, the quality and our assumptions will suffer from it. Despite the fact that parallelization gives in the first iteration almost 4 times speedup, all approximations have to be generated beforehand for the streaming purposes.

Appendix A

Examples of simplification

The examples below present the simplification of a mesh generated from a dense point cloud scanned in the NavVis office. The mesh has 507277 vertices and 973168 faces.

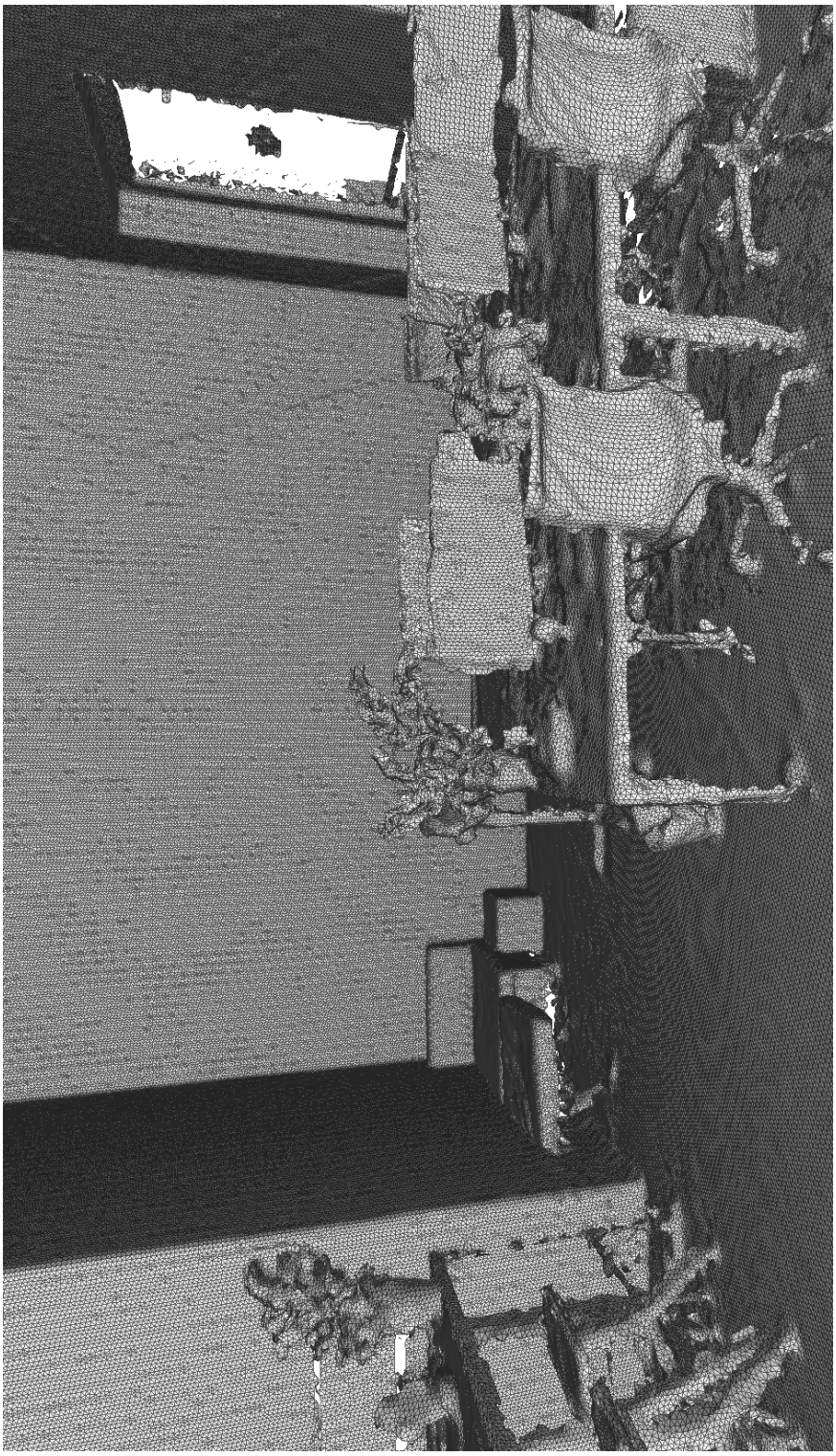


Figure A.1: Original mesh with evenly distributed triangles.

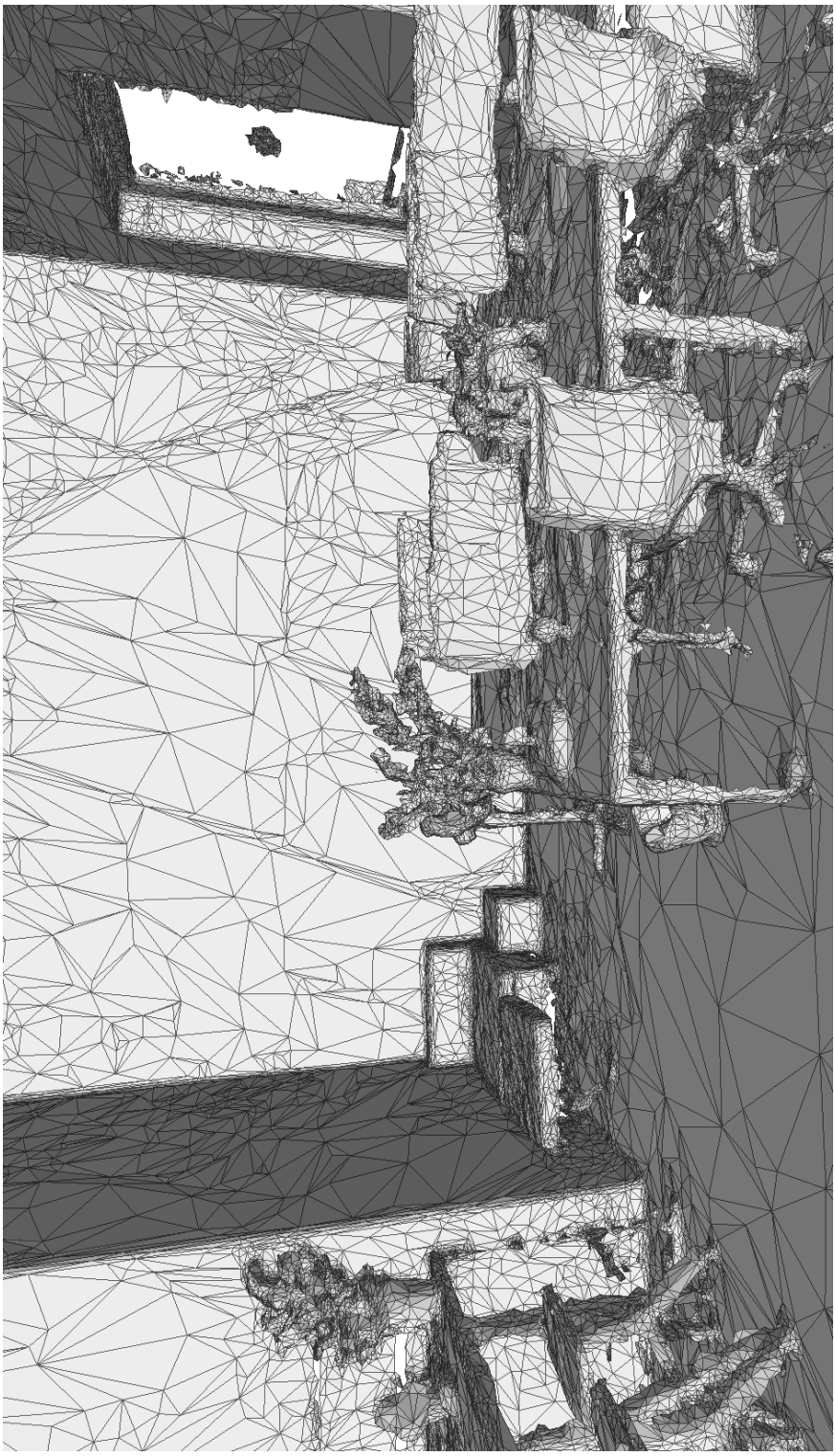


Figure A.2: Simplified mesh to 15% of the original using [geometry]

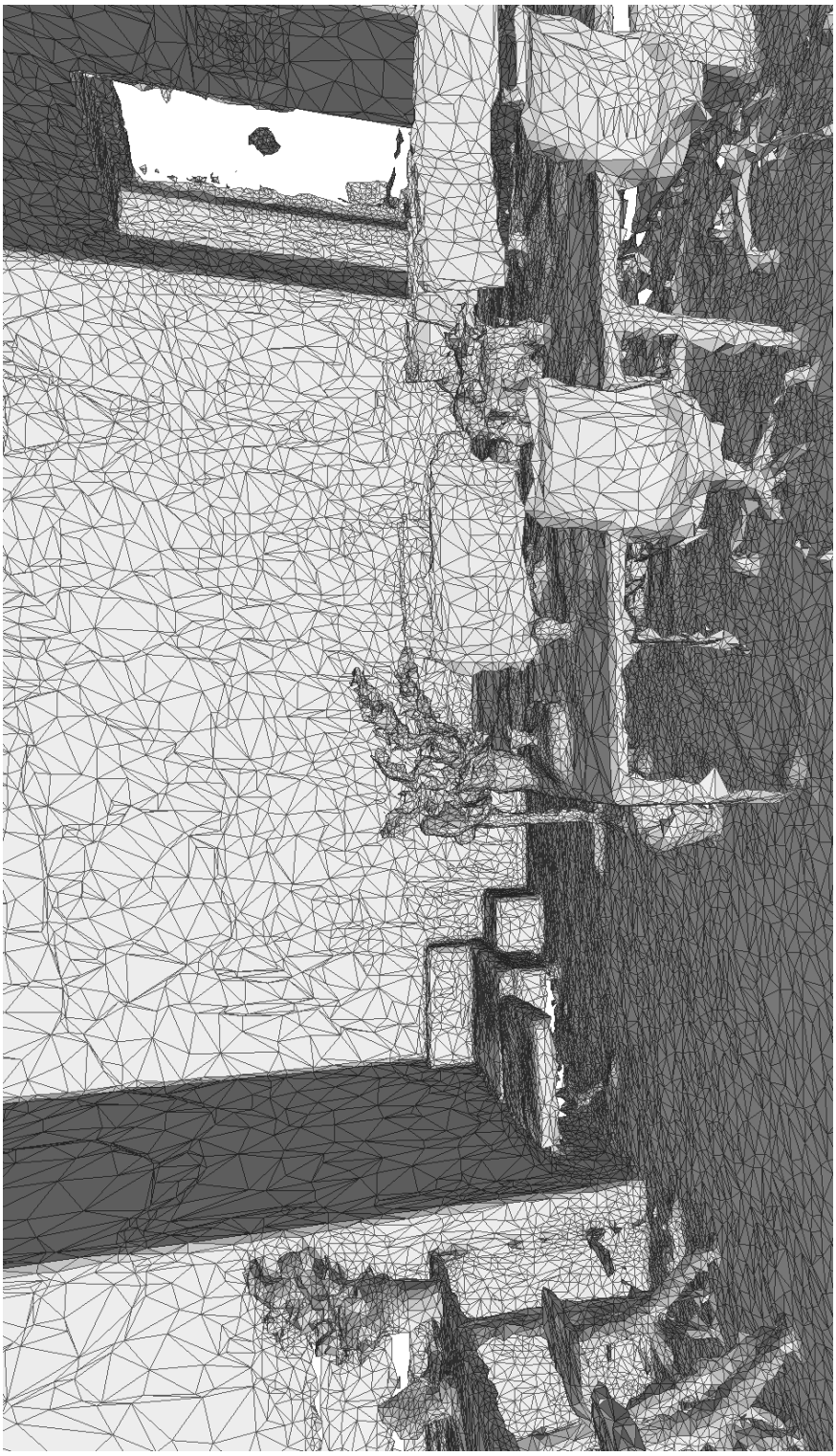


Figure A.3: Simplified mesh to 15% of the original using [geometry, color]

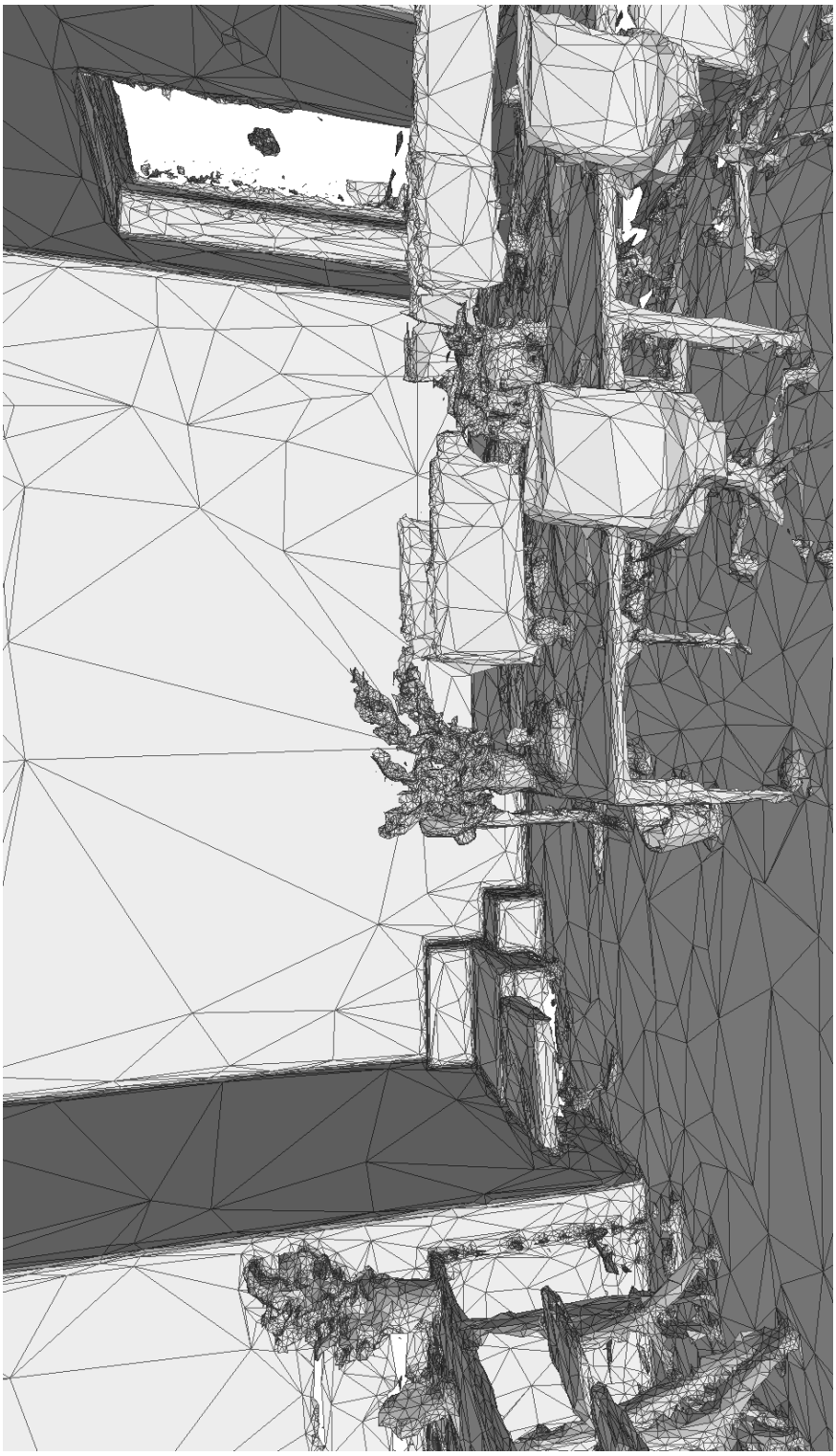


Figure A.4: Simplified mesh to 15% of the original using [geometry, color, normal]

List of Figures

List of Tables

Bibliography

[Amd67] Gene M. Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. ACM, 1967.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[Eig19] Eigen. `https://eigen.tuxfamily.org/doxclassEigen_1_1FullPivLU.html`, 2019. [Online; accessed 23-Oct-2019].

[Gar99] Michael Garland. *Quadric-Based Polygonal Surface Simplification*. School of Computer Science Carnegie Mellon University, 1999.

[GH97] Michael Garland and Paul S. Heckbert. *Surface Simplification Using Quadric Error Metrics*. ACM Press/Addison-Wesley Publishing Co., 1997.

[Gra02] Mark Grand. *Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML*. Wiley, Second edition, 2002.

[LRC+03] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., 2003.

[RT17] Juan Ropero and Gabriel Tamura. *Characterizing the Impact of Context-Variables in Software Performance Factors: a Domain-Specific Design Patterns Perspective*. 2017.

[Str88] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, San Diego, Third edition, 1988.

[SY01] C. C Jay Kuo Sheng Yang, Chang-Su Kim. *View-dependent progressive mesh coding for graphic streaming*. Multimedia Systems and Applications IV, 2001.

[Tau95] G. Taubin. *Curve and Surface Smoothing Without Shrinkage*. IEEE Computer Society, 1995.

[WE19] Kempf William E. `https://www.boost.org/doc/libs/1_33_1/doc/html/thread_group.html`, 2019. [Online; accessed 29-Oct-2019].