Surface Simplification

Topological Data Analysis Project Report

Gregor Kovač, Matic Šutar

January 2023

1 Introduction

We could say that surface simplification is the process of removing "details" from surfaces. It is used in different fields, most prominently in computer graphics. When we are rendering a scene we don't need all of the details of objects that are far away from the viewer, hence we can simplify it to decrease the rendering time and also reduce its file size. In this report we will present how we implemented surface simplification and also give insight into the produced results. The source code of this project can be found at **https://github.com/gregorkovac/surface-simplification**.

2 Methods

In this section we will talk about how we approached surface simplification and we will also give some information about our implementation. We will be working with triangulated 2-manifolds without a boundary.

2.1 Edge contraction algorithm

We used the *edge contraction algorithm*, that was presented to us in the book [1], for surface simplification. An edge contraction is the operation of replacing an edge of a surface with a new vertex and modifying the triangulation around it. A representation of this can be seen in figure 1. The algorithm takes all of the edges of a triangulation and sorts them by the error they would cause to the shape with contraction. Error assessment is explained more in detail later on, in section 2.1.1. Before contracting an edge we also have to check the *link condition lemma*, that we will take a look at in section 2.1.1. We then start contracting edges. We can either stop when we reach some error threshold or we can say that we want to contract a certain number of edges. We have opted for a combination of both, contracting N edges and also skipping those that cause too much error.

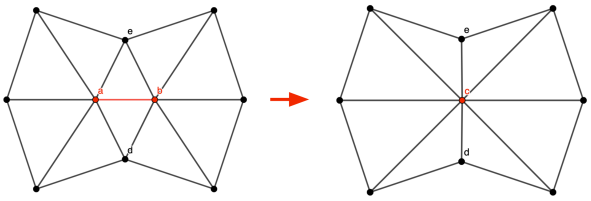

Figure 1: Contraction of an edge

2.1.1 Link condition lemma

The link condition lemma states that two surfaces K and L have the same topological type iff $\operatorname{Lk} ab = \operatorname{Lk} a \cap \operatorname{Lk} b$ where ab is an edge in K and L is obtained by contracting ab. The additional condition for this lemma to hold is that K is a 2-manifold without boundary. This essentially means that this will not hold on surfaces with holes.

2.1.2 Error assessment

In order to sort edges by the amount of deformation they cause to the triangulation, we need to define some sort of an error measure. Let us introduce the following function:

$$E_H(x) = \sum_{h_i \in H} d^2(x, h_i),$$

where h_i is a plane spanned by the i-th triangle of the surface, H is the set of all planes, x is a point and $d(x, h_i)$ is the distance of point x from the plane h_i. For purposes of later explanations we should also mention that a plane h_i is defined with its unit normal $u_i \in \mathbb{S}^2$ and offset δ_i. We can write h_i as a set of points $y \in \mathbb{R}^3$ for which $\langle y, u_i \rangle = -\delta_i$.

E_H will serve as our error measure. When contracting an edge ab the error is defined as:

$$\textbf{Error } ab = \min_{c \in \mathbb{R}^3} E_H(c).$$

We can search for the result using gradient descent. If we decide to contract ab, c will become the new point of the triangulation.

Using the above definition of $E_H(x)$ might not be the best performance-wise.

Therefore we can rewrite the function as follows:

$$\begin{aligned} E_H(x) &= \sum_{h_i \in H} d^2(x, h_i) \\ &= \sum_{h_i \in H} (\mathbf{x}^T \cdot \mathbf{u}_i)(\mathbf{u}_i^T \cdot \mathbf{x}) \\ &= \mathbf{x}^T \cdot \left(\sum_{h_i \in H} \mathbf{u}_i^T \cdot \mathbf{u}_i \right) \cdot \mathbf{x}, \end{aligned}$$

where $\mathbf{x}^T = (x^T, 1)$ and $\mathbf{u}_i^T = (u_i^T, \delta_i)$. We have extended both to four dimensions. Now we can further simplify:

$$E_H(x) = \mathbf{x}^T \cdot \mathbf{Q} \cdot \mathbf{x}.$$

We call $\mathbf{Q}$ the *fundamental quadric* of E_H. It is a four by four matrix:

$$\mathbf{Q} = \begin{bmatrix} A & P & Q & U \\ P & B & R & V \\ Q & R & C & W \\ U & V & W & Z \end{bmatrix}.$$

Writing $x^T = (x_1, x_2, x_3)$ we get:

$$E_H(x) = Ax_1^2 + Bx_2^2 + Cx_3^2 + 2(Px_1x_2 + Qx_1x_3 + Rx_2x_3) + 2(Ux_1 + Vx_2 + Wx_3) + Z.$$

Since we now have isolated the planes in the equation from the points, we only have to compute $\mathbf{Q}$ once when optimizing the error. However, even in this form it is quite expensive to compute $\mathbf{Q}$ over and over again. Luckily, it has some nice properties that will let us easily modify it throughout the algorithm. We start by computing quadrics of triangles $\mathbf{Q}_{abx}$ that are defined by the plane of the triangle. We can compute quadrics of edges with quadrics of triangles that contain that edge, like $\mathbf{Q}_{ab} = \mathbf{Q}_{abx} + \mathbf{Q}_{aby}$. Similarly, we can compute the quadric of a vertex by summing the quadrics of triangles that contain it.

Because triangles share edges and vertices, we can introduce a relationship between sets of planes. For a triangle abx with edges ab, bx, ax and vertices a, b and x it holds that $H_{ab} = H_a \cap H_b$ and $H_{abx} = H_{ax} \cap H_{bx}$. Now we can say that when contracting an edge ab we can compute the new quadrics like

$$\begin{aligned} \mathbf{Q}_c &= \mathbf{Q}_a + \mathbf{Q}_b - \mathbf{Q}_{ab} \\ \mathbf{Q}_{cx} &= \mathbf{Q}_{ax} + \mathbf{Q}_{bx} - \mathbf{Q}_{abx} \\ \mathbf{Q}_{cy} &= \mathbf{Q}_{ay} + \mathbf{Q}_{by} - \mathbf{Q}_{aby}, \end{aligned}$$

where c is the new vertex and cx and cy are new edges. This is much more efficient compared to computing the new quadrics from scratch.

2.2 Implementation

For an efficient implementation of the described algorithm we used different data structures and an optimization technique that was described in the book [1].

We abstracted vertices, edges and triangles as simplices of varying degrees and represented them with a class labeled as `Simplex`. The class implements basic functions related to simplices, such as accesing neighbouring simplices, computing the link of a simplex and others. Objects of this class were used as the basic elements of other data structures.

For the purpose of accessing neighbouring simplices in constant time (relative to the size of the triangulation), we implemented the *Hasse diagram*. A Hasse diagram is in essence a collection of simplices that form a desired simplicial complex. Every simplex is then connected to (can access) all of its facets ($(n-1)$-faces) and cofaces (as in simplices that are of one degree higher than the simplex in focus and also contain the simplex in focus). The only non-trivial step of our specific implementation was the removal of the simplices when contracting an edge.

The second data structure we implemented was a *MinHeap* and was used as a priority queue for sorting edges according to the error measure. We implemented the structure in a standard way so that the heap is an array representing a binary tree. The heap is efficiently initialized using *sift up* operations. Since the algorithm could potentially update the error measure of any edge, we also had to store the information about the location of each edge in the heap and update it when needed.

Last but not least, we used an optimization technique for calculating the fundamental quadrics of simplices. In the book [1] the technique was called "maintenance of the error measure", and we also described it in the chapter 2.1.2. The main task for successfully implementing the technique was to correctly identify the neighbourhood that needed to be updated when a contraction of an edge occurred.

One detail we would like to emphasize is that we didn't tackle the cases where the model was not a 2-manifold without a boundary. This means that the algorithm will not work completely correctly on surfaces that, for example, contain holes (the topology type might change in the process of simplification).

3 Results

In this section we will present the results of our implementation in the form of rendered images. We will also assess the time complexity of the algorithm and we will say a few words about the homology of simplified surfaces.

3.1 Renders

We exported our simplified surfaces as *object* files and created the renders in the 3D modeling software *Blender* with the use of the *Cycles* renderer.

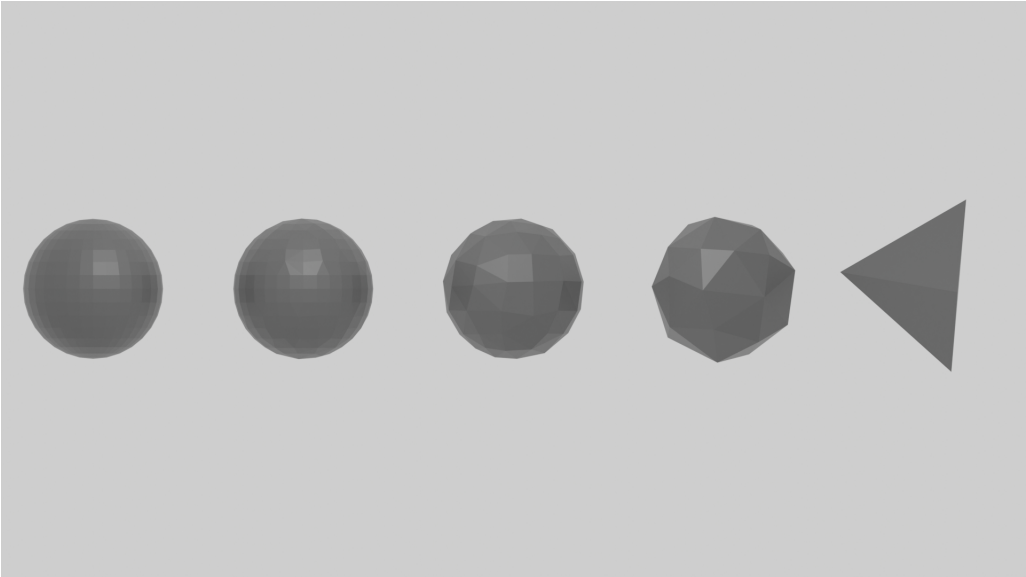

Figure 2: A sphere at different simplification levels

First example render is of a sphere, that we can see on figure 2. We can see that if we simplify it as much as possible, we get a tetrahedron.

The following two examples use models which were taken from **the Stanford 3D Scanning Repository** (*https://graphics.stanford.edu/data/3Dscanrep/*). The first one is the bunny and it can be seen in figure 3. On the first resolution level we can see that if the error threshold is too low, a surface can get very deformed, but it still preserves at least a general shape. One thing about the bunny is that it has a few holes, which we said before is not good. On the third resolution level we can see that a major deformation is caused near the feet of the bunny because of these holes. For the full resolution we first used *Blender* to fill in the holes and now we can see that we get a very nice simplified model without major deformations. We had to do around 34 000 edge contractions to reach the most simplified version of the full resolution model. For example for the lowest resolution we only used 500 contractions and it already caused big deformations.

We have also tried the algorithm on 4. We had to do around 22 000 edge contractions to reach the most simplified version.

3.2 Algorithm complexity

We will first introduce some basic notation. Let $O(\cdot)$ represent the *big O notation*, let T be the number of triangles in the initial triangulation (note that for 2-manifolds with no boundary, $O(E) = O(T)$, where E is the number of edges in the initial triangulation) and let n be the number of contracted edges. Let's

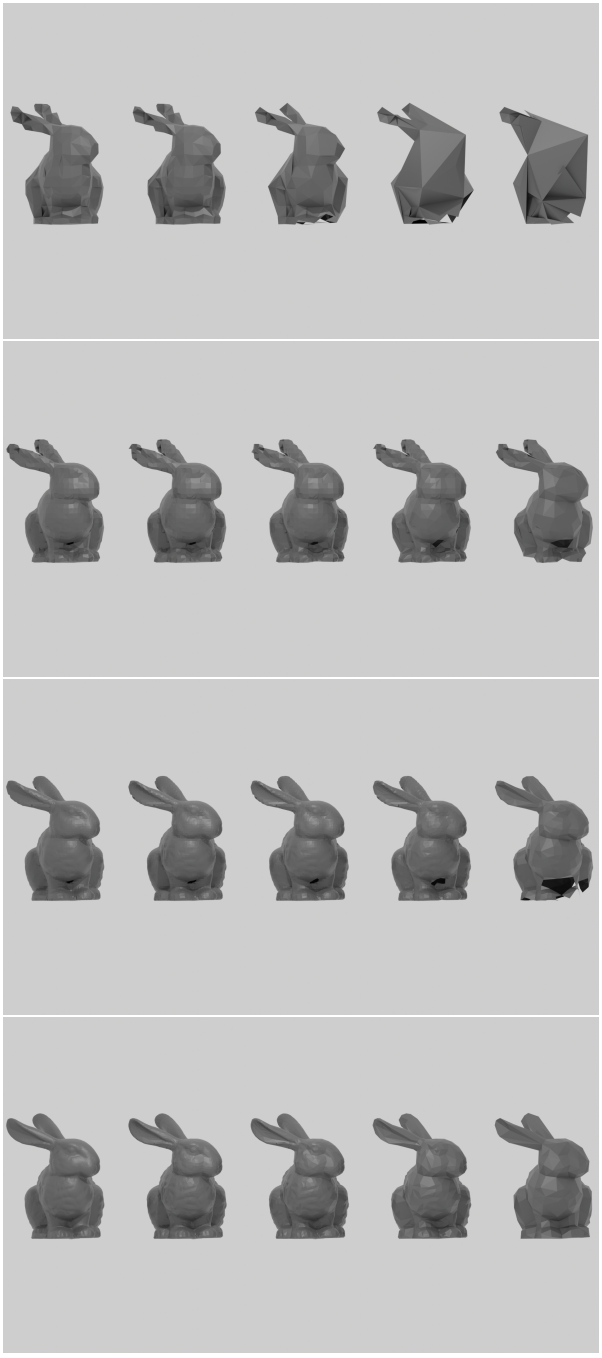

Figure 3: The Stanford bunny of all four resolution levels (top one is smallest resolution and the bottom one is full resolution) at different simplification levels. It can be seen, that the algorithm starts to break when simplifying models with holes (lemma 2.1.1 doesn't hold).

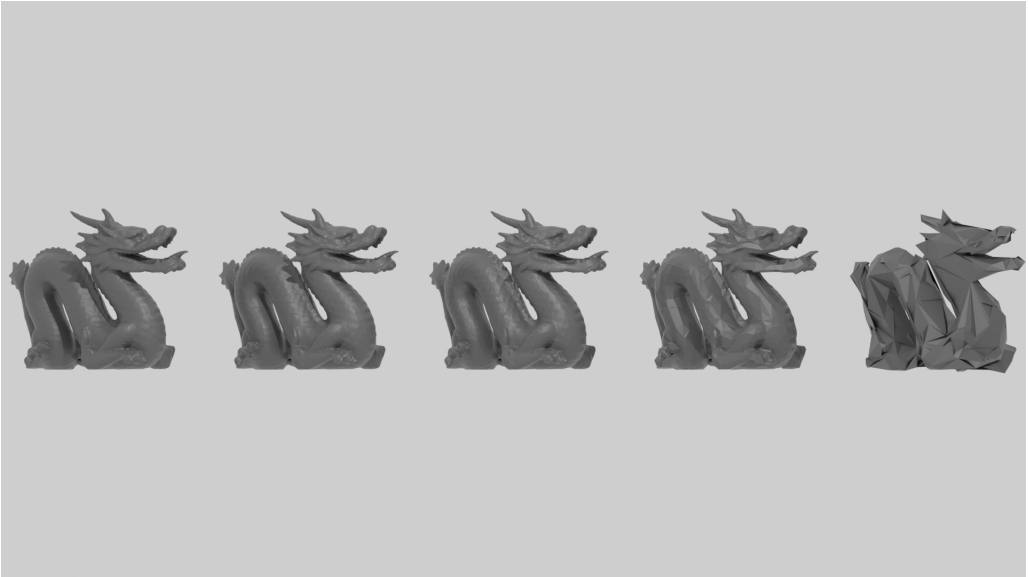

Figure 4: The Stanford dragon at different simplification levels.

now identify all the major operations that take place during the execution of the algorithm.

The algorithm starts by initializing the Hasse diagram. The initialization of Hasse diagram has a complexity of $O(T)$. The second step is to initialize all the fundamental quadrics and errors which has again a complexity of $O(T)$ (with a relatively large constant factor, mostly due to the calculation of error, specifically the calculation of the position of the new vertex). After the errors are computed, the algorithms sorts the edges by building a heap. Since the heap is initialized using the sift up operations, it has a time complexity of $O(T)$. The main loop consists of popping an edge from the queue (heap), checking the link condition and of the contraction of an edge. The operations have time complexity of $O(\log T)$, $O(1)$ and $O(\log T)$ respectfully. Contraction of an edge consists of contracting an edge in the Hasse diagram ($O(1)$), removing two edges from the queue ($2 \cdot O(\log T)$), updating quadrics of neighbouring simplices ($O(1)$), updating errors of edges in the neighbourhood ($O(1)$) and updating the edges in the priority queue ($k \cdot O(\log T)$, where k is some constant and is not dependant on T).

Putting all the parts together, the time complexity of the implemented simplification algorithm is $O(T+T+T+n\cdot(\log T+1+\log T)) = O(3T+n\cdot(2\log T+1)) = O(T + n \cdot \log T)$. It is important to note that this is the asymptotic time complexity which hides a relatively large constant factor which in practice plays an important role in the actual execution time. For this reason and because of the fact that we implemented the algorithm in *Python*, the execution time for the larger models is in order of minutes.

3.3 Preservation of homology

When it comes to homology of the original and simplified surfaces we can say that it is preserved with simplification. If we do the simplification correctly, we know that it won't tear or glue the surface, close holes or open them, hence it is a homeomorphism. We know that homology is preserved under homeomorphisn.

We have also tested this by computing the homology of the surfaces. For example for the full resolution Stanford bunny (with filled holes) we get the following homology groups:

$$H_0 = 1, H_1 = 0, H_2 = 1.$$

For the final simplified version we got the same result. We also tested this on the Stanford dragon and on the sphere, where the homology is also preserved. We can conclude that our assumption about the preservation of homology is true.

4 Discussion

We have successfully implemented the surface simplification algorithm presented in [1] and optimized it so that it runs relatively fast. We have presented our results on some examples and created nice looking renders where we can clearly see the effect of the algorithm. We have also assessed the time complexity and talked about the preservation of homology, which is a nice property of this algorithm. We are satisfied with our work. The only further improvements might be even more optimization to hopefully make it execute in real time and handling of the cases where the input surface is not a 2-manifold without boundary.

Division of work

Gregor implemented the basic functions of the algorithm, calculated homology and created the model renders. Matic finished the algorithm, rewrote the code structure and optimized everything.

References

[1] *Computational topology: an introduction.* Edelsbrunner, Herbert and Harer, John L. 2022, American Mathematical Society.