Minimal Surfaces from Triangular Meshes

Automatically Converging to Smooth Versions of Triangular Meshes

Julian Vos

A Thesis Presented for the Degree of Bachelor of Science
Supervised by Jiří Kosinka and Pieter Barendrecht
Faculty of Mathematics and Natural Sciences
University of Groningen
July 1st, 2016

Minimal Surfaces from Triangular Meshes

Automatically Converging to Smooth Versions of Triangular Meshes

Abstract

In this thesis I investigate the novel use of the centroidal Voronoi tessellation (CVT) concept as a surface mesh smoothing operation. I carry out extensive literature research to find the most promising way for implementing the computation of such tessellations on 2-manifold triangular meshes embedded in three-dimensional space. I detail my mesh visualisation application which implements this operator, as well as Loop subdivision, Laplacian smoothing, and dual mesh generation. I perform experimental research to find out how the new smoothing operator can best be combined with these other three, more traditional ones. For this purpose I focus on open meshes, so that boundary vertices can be kept fixed in order to obtain a smooth mesh surface between the boundaries. I search for the optimal strategy of converging to surfaces that still resemble the input mesh while appearing smooth to the human observer, without requiring too high a mesh density or too long a computation time. I report my findings in a coherent conclusion, and specify remaining future work on the subject.

Contents

1 Introduction

Three-dimensional objects are often represented digitally by triangular meshes. Meshes, which are approximations by nature, conveniently allow for a modeller and for computer programs to examine a model's structure, and to alter it by moving its vertices, changing edge connectivity, and adding/removing vertices, edges, and faces as seen fit, without requiring much storage space. Their mathematical properties, like normals, are also easy to calculate, and hence displaying them is simple, certainly when it concerns triangular meshes. They are therefore widely supported by major computer graphics software, like `OpenGL`, and can be seen as the industry standard for representing three-dimensional objects.

Meshes can be modelled directly, or they can be computed by an algorithm, sometimes based on actual scanning data of an object. They are becoming even more important with the current rise of 3D printers, as printable files usually take the form of a triangular mesh. Despite being approximations, meshes can accurately represent smooth surfaces when they are dense enough, and such representation quality is important to many humans and applications.

However, there is a balance to be found here, certainly when it concerns manual modelling. A mesh with a low amount of vertices is made quickly, and is relatively inexpensive to render or print. However, the mesh surface will appear jagged to the human eye. Increasing the amount of vertices, and correspondingly edges and faces, will result in a smoother surface, but at the cost of more modelling and rendering time. What is needed then is an automatic, robust way of smoothing an initial triangular mesh, such that the amount of vertices is kept as low as feasible without sacrificing a smooth looking surface.

To provide for this, I introduce and test four different smoothing operators, which all take a triangular mesh as input, and output a slightly smoother version of it: 'Loop subdivision', 'Laplacian smoothing', 'generate dual', and most importantly, 'generate centroidal Voronoi tessellation'. Whereas Loop subdivision systematically adds more vertices to a mesh, the other three operators take care of redistributing vertices based on their relative positions in three-dimensional space. Note that the generate dual operator will always have to be applied twice in order to get back to a triangular mesh, so that the other operators remain applicable, as I explain in the next section. Later in the thesis I accordingly refer to this operator as 'generate dual-dual'.

Out of the four operators included in this research, generate centroidal Voronoi tessellation is the only one that changes edge connectivity after vertex redistribution. As I show, this is very important to ensure high mesh (triangle) quality while the smoothing process converges to a stable state, a factor which not only allows for more convenient post-processing, but which also increases the perceived smoothness of the model without introducing additional computational overhead. The closer triangles are to being equilateral, the better.

To carry out my research, I start out from an existing application for performing Loop subdivision, as supplied by P. Barendrecht from the University of Groningen, modify it to suit my needs, add implementations of the other three smoothing operators, and use the resulting application to obtain results.

1.1 Scientific Contribution

For this thesis, my research question is the following: "Given an initial mesh, can the operators of Loop subdivision, Laplacian smoothing, generate dual-dual, and generate centroidal Voronoi tessellation together converge to a smooth variant of high quality when boundaries are kept fixed, and if so, in what order to achieve the best result?". Note that boundaries shall be kept fixed, because otherwise every mesh converges to a single point, thwarting the goal of automatically and robustly turning a coarse initial mesh into a resembling variant that appears smooth to the human eye without becoming more dense than necessary. As only open meshes have boundaries, I concentrate on those (see Section 2).

Such smooth but thin surfaces between fixed boundaries are close to the mathematical concept of 'minimal surfaces', which are surfaces with a mean curvature of zero, and which can be seen as 'minimised surface area subject to a volume constraint'. Another way of stating the research question then, although less accurate, is: "How to obtain minimal surfaces from triangular meshes?".

As has been proven recently, minimal surfaces are related to the concept of centroidal Voronoi tessellations [1], indicating the promising prospects of generate centroidal Voronoi tessellation as a smoothing operator. This use is novel, and therefore this operator shall be the focus of my thesis. More extensively so than for the other, more traditional operators mentioned, I discuss possible ways for implementing it and look into its convergence properties by examining the literature. I perform various experiments to find out how this operator can best be combined with the others to achieve the smoothest stable result.

1.2 Thesis Format

In the remainder of this thesis I clarify the concepts mentioned so far, examine the literature on centroidal Voronoi tessellations, discuss my own application's details, and present experimental results to then answer the research question. In Section 2 I explain further the concepts of closed and open triangular meshes, and of the four used smoothing operators. In Section 3 I summarise the existing literature about the applications, algorithms, and convergence properties of centroidal Voronoi tessellations, and their relation to minimal surfaces and my subject, so that I am able to implement this operator properly. In Section 4 I illustrate my research approach with relevant implementation details. In Section 5 I present obtained results for both simple and complex cases. Finally, in Section 6, I provide a conclusion to the thesis, answering the research question and shortly naming possible future extensions and research subjects.

2 Concepts

As specified in the above, I use triangular meshes, some closed but mainly open ones, to smooth by means of four different but somewhat similar smoothing operators, which I apply in varying order to gain research results. Before I proceed, I first clarify these important concepts further so that the remainder is more easily understood. First I work out the notion of (triangular) surface meshes, then I define the difference between closed and open ones, and finally I explain the four operators one after another, mentioning both the two-dimensional case and, more extensively, the three-dimensional case that I am interested in.

2.1 Triangular Meshes

A surface mesh is a collection of faces, edges, and vertices, that together surround some volume that they, with another word, represent. Vertices are points, edges are connections between such points, and faces are polygons (as opposed to polyhedra, as would be the case for a volumetric mesh) bounded by edges. For a triangular mesh, these polygons are all triangles, and thus consist of exactly three edges and vertices (Figure 1). Being the most simple, most common, and easiest processable kind of mesh, they form the subject of my research.

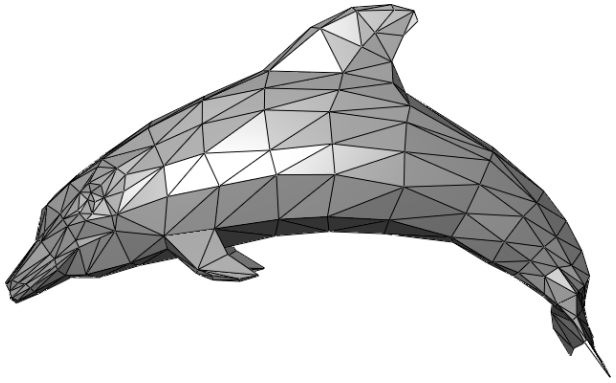

Figure 1: An example triangular mesh, enclosing volume representing a dolphin. (courtesy of Wikimedia)

2.1.1 Closed and Open Meshes

There is a clear distinction to be made between closed and open meshes however. The faces of a closed mesh fully enclose a volume, without leaving any gaps. On the other hand, the volume corresponding to an open mesh is only partly enclosed by its faces, and one or more surface gaps are present. These gaps are encircled by so-called 'boundary loops'; loops of edges which surround the gaps. In other words, open meshes have boundaries whereas closed meshes do not.

Of course in both cases the represented volume is bounded by faces, and so the volume does always have 'boundaries' in the form of the mesh. But for open meshes this mesh surface is incomplete; it is the open mesh surface itself which has boundaries as opposed to a closed mesh surface, which has none (Figure 2).

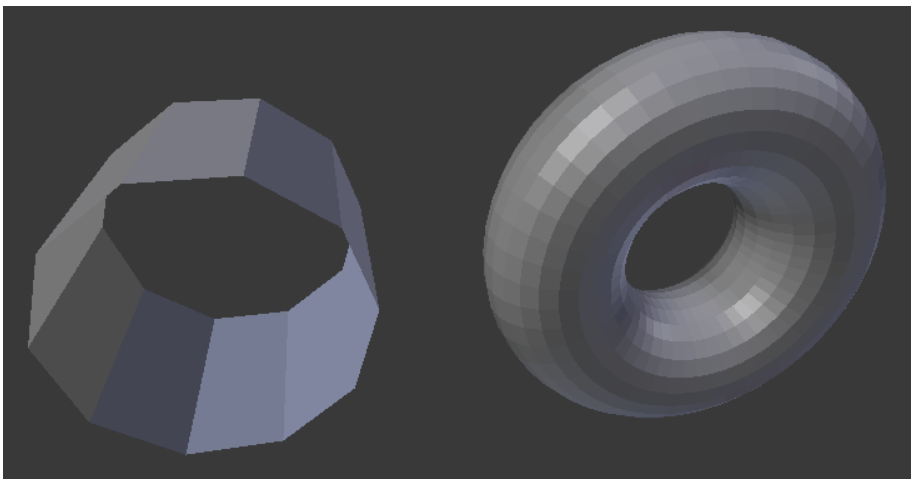

Figure 2: An open cylinder mesh and a closed (no surface gaps) torus mesh. (visualised using `Blender`)

In the following I use the word 'boundary' to refer to a surface boundary, as part of a boundary loop. Vertices connected by edges that constitute the boundary loops are called boundary vertices. As stated before, open meshes are more interesting to me than closed meshes, because closed meshes eventually converge to a single point which is the average of all original vertex positions. For the Laplacian smoothing, generate dual, and generate centroidal Voronoi tessellation operators to reach a stable smooth state then, some vertices have to remain fixed, and boundary vertices provide a great solution to that.

I fix boundary vertices in place, except when it comes to Loop subdivision. This operator is not meant for redistributing vertices, but mainly for increasing the amount of vertices. Still, I allow this operation to slightly alter the positions of boundary vertices, because it does not work properly otherwise, and because this has proven not to be a problem for my research.

A drawback of this decision is that generate dual-dual can only output logical and useful results for closed meshes. As will be clear from the next subsection, this operator, in its general implementation, removes all boundary vertices after every appliance, creating new boundary vertices until the gaps have consumed the mesh. For the largest part of the experimental research section (5) I therefore focus on the other three operators, treating closed meshes only briefly for the sake of completeness. Still, for smoothing closed meshes the generate dual-dual operator is handy, just not up to convergence, which is a property I am after.

2.2 Available Smoothing Operators

Now I shortly illustrate all four operators, with a focus on the generation of centroidal Voronoi tessellations, before moving on to the literature section which expands on that particular operation. Loop subdivision receives somewhat less extensive coverage here, because I did not implement this algorithm myself and it thus forms a smaller part of my scientific contribution.

2.2.1 Loop Subdivision

As mentioned, Loop subdivision increases the amount of vertices of a triangular mesh, thereby allowing a smoother representation of an object. What it does is dividing up every triangular face in four smaller triangles, using quartic box splines to calculate the new vertex positions in order to obtain new faces under smooth angles (Figure 3). In other words, for each face three new vertices and edges are added to the mesh as it is split up into four similar faces.

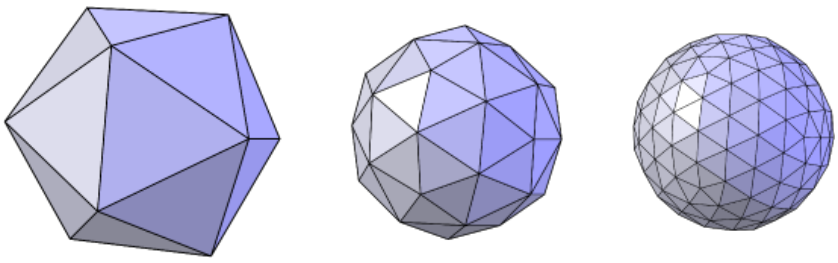

Figure 3: Loop Subdivision of an icosahedron, after 0, 1, and 2 steps. (courtesy of Wikimedia)

In theory, Loop subdivision can be applied infinitely many times to approach a smooth limit, but in practice an average computer cannot handle more than five subdivision steps for a relatively small initial mesh without the computer struggling to handle the greatly increased amount of data. Hence I restrict its usage insofar this is necessary for my laptop to keep running smoothly.

2.2.2 Laplacian Smoothing

Laplacian smoothing is the most straightforward algorithm for redistributing vertices based on their relative positions. Every vertex is simply moved to the average position of its neighbours. The word 'neighbour' is understood here as another vertex linked to the vertex under consideration with an edge, disregarding distances. So, the term is about topology, not about geometry.

A vertex with 5 outgoing edges has 5 neighbours, and when applying one step of Laplacian smoothing to the mesh it relocates to their average position. As the same operation is applied to each vertex simultaneously, the resulting mesh is a smoother one (Figure 4). Note that this operator does not change anything about edge connectivity and faces; it merely smoothens geometry.

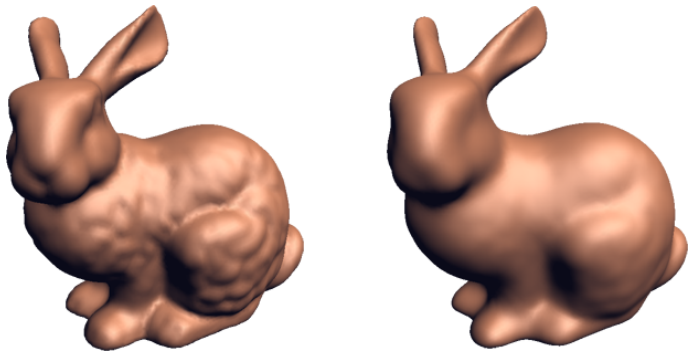

Figure 4: Repeated Laplacian smoothing of the Stanford bunny test model. (courtesy of Caltech's Introduction to Discrete Differential Geometry)

Like generate dual-dual and generate centroidal Voronoi tessellation, Laplacian smoothing only takes into account local information, rendering it inexpensive. Being the simplest of the three, it is the fastest to apply, and like generate dual-dual, its closed mesh limit is a relocation of all vertices to the initial mesh's barycentre. Luckily, this is not the case for open meshes, as long as boundaries are kept fixed. In that case, although usually after hundreds of steps, the operator tends to converge to a stable state where all vertices are located at their neighbours' average and hence stop moving.

To add a bit more versatility to this operator, I add to it a parameter w, which is to be seen as the smoothing weight. When $w = 1$, vertices are be moved to the average position of their neighbours. When $w = 0.5$, they re only moved halfway there. When $w = 0$, no change (movement) occurs at all. Thus, the Laplacian smoothing weight w specifies how far the vertices are moved towards those average neighbour positions; in other words, it controls the strength of the smoothing. As the mesh, for degenerate cases of $w = 1$, can theoretically get stuck in a loop that goes back and forth between two vertex distributions, I default to $w = 0.5$ for the remainder of this thesis. But note that my application, associated with the thesis, can be used to experiment with other values for w.

2.2.3 Generate Dual Mesh

Generating a dual mesh twice is similar to applying a step of Laplacian smoothing, in the sense that vertices are moved towards their neighbours. However, dual meshes are more complicated because they involve faces and information about their adjacency, and edge connectivity actually has to be updated. Nonetheless, after two steps, or in other words, after applying the generate dual-dual operator, edges and faces are back to their original state, and only vertex positions are different. To clarify this, I first explain what it means to generate a dual for a mesh, illustrated with a two-dimensional example (Figure 5), before explaining what the effects are of generating the dual of a dual mesh, and how this concept can easily be extended to three dimensions.

The generate dual operator creates a new mesh which is said to be dual to the current mesh. Every current face gets a new vertex in the middle of it, and afterwards all new vertices are connected by edges to those other new vertices that correspond to adjacent faces of the current mesh. What happens then is that all current vertices get a new face around them, as can be seen in Figure 5 below: the blue mesh is the current one, and the red mesh the new dual.

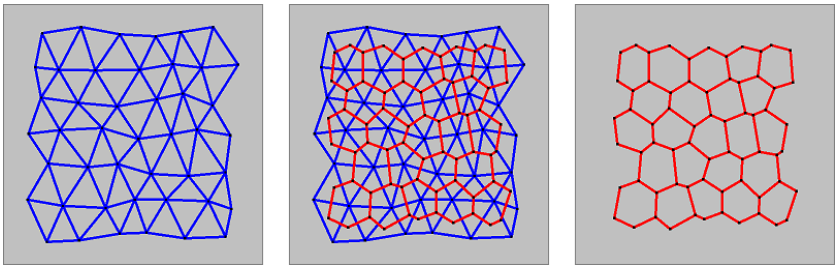

Figure 5: A mesh, the mesh overlayed with its dual mesh, and the dual itself. (from a 2D application I wrote myself for exploratory purposes)

What is left to decide on is how to compute 'in the middle of it'. There are two logical options: placing new vertices at faces' centroids, or at their circumcentres. A face's centroid is its centre of mass. As I work with abstract meshes that do not hold vertex, edge, and face mass information, I simply take this to be the barycentre of a face's vertices. The circumcentre of a face is that point which has an equal distance to all its vertices. Not only is this more complex to compute, it also does not usually exist for non-triangular polygons. Hence, as a first dual creates non-triangular polygons and I need to be able to apply the generate dual operator again, I stick to centroids.

The dual of a triangular mesh is non-triangular, and for a regular mesh in fact precisely hexagonal. But the second dual, or the dual of the dual, is again triangular. Because every original face gets a vertex at its centroid, and then again a face around that vertex, the original topology is restored. Thereby the mesh becomes a smoothed version of itself. To use this effect properly I, as stated before, combine this into one, convenient operator: generate dual-dual.

Like Laplacian smoothing, generate dual-dual is logically and easily generalised to three dimensions, because it needs to know only topological information. On the other hand, also in the three-dimensional case the operator still only works for closed meshes, as it assumes vertices and faces to be fully surrounded by adjacent faces. Extending it to open meshes is not trivial, and would not make much sense in the pursuit of my research contribution.

2.2.4 Generate Centroidal Voronoi Tessellation (CVT)

Although my fourth and final smoothing operator is similar to the previous two in that its main use is redistribution of vertices, it differs in the sense that it is mostly a geometric operator: rather than edge connectivity, it uses the actual

distance between vertices to relocate them. After that it even flips edges, which is an important difference as well. This way it cannot only smooth the triangular mesh better, but also ensure high quality of its output.

The most complicated, computationally expensive, and untested of the operators, I spend more time handling it than on any other, for a good part in the form of the next section on centroidal Voronoi tessellations in the scientific literature. Given its effects, the smoothing prospects are promising. First however, I describe the general idea for two dimensions, and end this section with a short discussion of possible options for extension to the three-dimensional case.

A centroidal Voronoi tessellation is a specific kind of Voronoi diagram, or Voronoi tessellation. A Voronoi diagram is a partitioning of a Euclidean space into a set of distinct Voronoi regions, or Voronoi cells, based on a finite set of 'generators', which are usually points, and in my case vertices. Each generator has its own corresponding cell, which consists of all points in the space which are closer to that generator than to any other. This way, there is a finite number of non-overlapping Voronoi cells equal to the amount of generators, and together they fill up the whole space. A centroidal Voronoi tessellation is a Voronoi diagram whose generators coincide with the centroids of the cells (Figure 6).

Centroidal Voronoi tessellations, or 'CVT's as they are often referred to, can be viewed as an optimal partition of space corresponding to an optimal distribution of generators. They can be constructed by means of Lloyd's algorithm, named after Stuart P. Lloyd [2]: compute the Voronoi diagram of a set of generators, calculate the cells' centroids (centres of mass/arithmetic means), move the generators to the centroids, and repeat this procedure until convergence. To make this work in practice, 'convergence' has to be understood as the difference between two iterations becoming smaller than a predefined tiny amount.

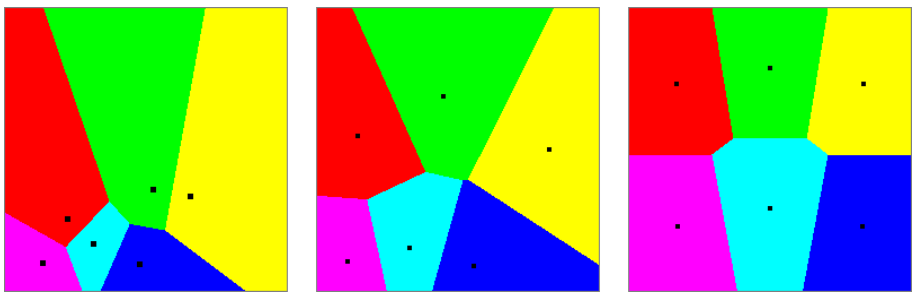

Figure 6: The initial Voronoi diagram, the diagram after one Lloyd step, and the converged CVT. (from another 2D application I wrote myself)

So, the generate centroidal Voronoi tessellation operator takes a mesh as input, and outputs a modified version of it with vertices relocated in such a way that the distances between any two neighbours become similar, as can be seen from the third image in Figure 6 above. Moreover, Voronoi cell adjacency information can be used to reconstruct and improve edge connectivity: if two cells are adjacent, connect their generators (vertices) with an edge.

The result is known as a 'Delaunay triangulation', which is a specific way of triangulating a set of points, that is associated with high triangle quality and which is dual to the points' Voronoi diagram (Figure 7). This kind of triangulation is formally defined as one for which no point lies inside the circumcircle of any triangle of the triangulation, maximising the minimum present angle.

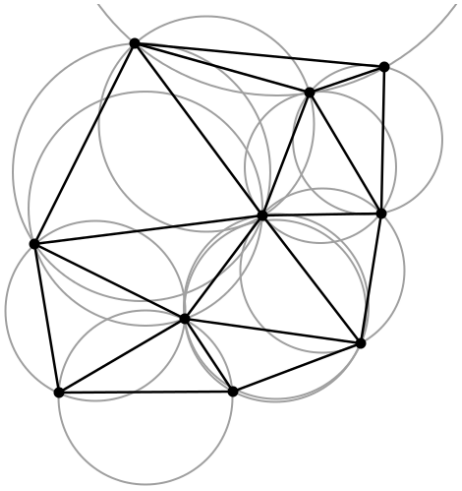

Figure 7: The unique Delaunay triangulation of 10 points in the plane, showing the triangles' empty circumcircles. (courtesy of Wikimedia)

As centroidal Voronoi tessellations do not only redistribute vertices nicely, but further smoothen the mesh's appearance by flipping edges, they have recently found many applications in science, technology, and even visual art [3]. Sometimes though, and certainly for my research subject, the operator has to be extended to three dimensions. This is not as straightforward as for Laplacian smoothing and generate dual-dual, now that geometry is being taken into account. A rough distinction is that between the regular three-dimensional case, which results in convex polyhedra for cells rather than convex polygons, and computing polygon cells on a surface mesh, which is a collection of 2-manifold surfaces (faces). The latter, though required in my case, happens to be more difficult. I delve into this in the following section, while also investigating ways of speeding up the linear convergence achieved by Lloyd's algorithm, as acceptable speed is vital to the success of a complex operator like 'generate CVT'.

3 Existing Work on CVTs

Recently, quite a lot of scientific articles have been written on the subject of centroidal Voronoi tessellations. This research interest is mainly due to their wide range of applications, including geometric modelling, image and data analysis, image compression, finite difference methods, numerical partial differential equations, quadrature, statistics, distribution of resources, cellular biology, and the territorial behaviour of animals [3, 4]. In my case however, I am interested in the CVT's inherent high quality remeshing ability.

Various algorithms for computing centroidal Voronoi tessellations have been devised, which vary in their application domain: some work only in two dimensions, others compute three-dimensional polyhedral Voronoi cells, and a few find polygonal cells on meshes, in other words, on 2-manifold surfaces embedded in three-dimensional space. Some of these approaches also constrain ('clip') the resulting diagram to a specific domain, often a square or cube. In general however, two main categories of approaches can be distinguished, both iterative in their own way: computing a Voronoi diagram every step and iteratively turning it into a centroidal tessellation, and minimising a CVT energy functional.

The first approach, iteratively computing Voronoi diagrams until this process converges to a centroidal Voronoi tessellation, comes down to Lloyd's algorithm, which I had already described shortly in Section 2. Further up in this section I describe it in more detail. The second approach, minimising a CVT energy functional, is more novel, and amenable to faster (superlinear) convergence, as I explain soon. This energy functional is understood as the sum or average of all Voronoi cells' integrals of their points' squared distance to their generator.

Naturally, when this function is minimised, the generator locations corresponding to a centroidal Voronoi tessellation have been found, which needs not be unique. If one wants to take these as mesh vertex positions, computing their Delaunay triangulation establishes the required edge connectivity. This process can be simplified if the input was itself a mesh, because then the old connectivity is known and edges only have to be flipped until the triangles satisfy the Delaunay property (empty circumcircle) again.

Delaunay triangulations are also often used to compute Voronoi diagrams, as a Voronoi diagram is dual to its generating points' Delaunay triangulation, as stated in the previous section, and they are easier to construct. I now first discuss ways known from the literature for Voronoi diagram computation in 2D, 3D, and on polygon meshes, before moving on to convergence properties, when I extensively cover the two indicated main approaches for computing CVTs, and shortly discuss multigrid methods, which can further speed up CVT computations. I end this section with elucidating the mentioned proven relation between centroidal Voronoi tessellations and minimal surfaces.

3.1 Voronoi Diagram Computation

Computing regular Voronoi diagrams of sets of points is worthwhile on its own, and likely needed to answer my research question, as most algorithms for com-

puting CVTs described in the scientific literature involve it in their iterations; even some that minimise an energy functional. Although I am ultimately interested in those algorithms that work directly on meshes, I first describe the methods that work in 2D only or in 3D as well, to provide a complete overview.

3.1.1 In Two Dimensions

A classic and efficient algorithm for computing a Voronoi diagram of a set of points in the plane, that is, in two dimensions, is 'Fortune's algorithm', invented in 1986 by S. Fortune [5]. Known as a sweep line algorithm, it propagates a straight line through the plane, with the generating points left of the plane incorporated into the Voronoi diagram already and the generators at its right still awaiting consideration (Figure 8). When the line has crossed the whole plane, the Voronoi diagram is finished. I do not go into the complex specifics, but note that this approach cannot be extended to three dimensions easily.

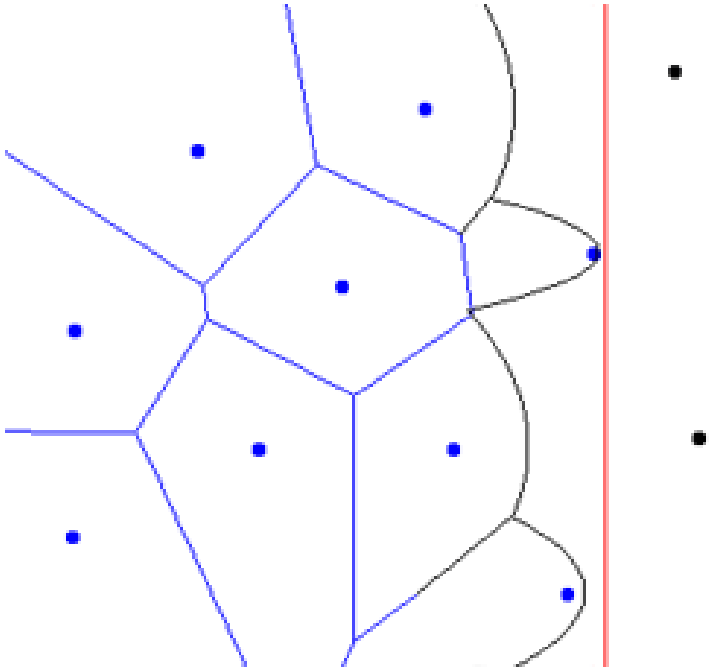

Figure 8: Fortune's algorithm during its propagation. (courtesy of Wikimedia)

Another classic, and nowadays more relevant algorithm for computing two-dimensional Voronoi diagrams is that of taking the dual of the generating points' Delaunay triangulation, which is itself found via edge flipping based on the empty circumcircle property. This approach was pioneered by L. Guibas and J. Stolfi in 1985 [6] and can, unlike Fortune's algorithm, be extended to three dimensions by considering empty circumspheres instead of circumcircles. Such extensions are still the de facto standard in 3D Voronoi diagram computation.

3.1.2 In Three Dimensions

The naive way for approximating a three-dimensional (or two-dimensional) Voronoi diagram is a fine grid discretisation; a discretisation of a space into fine grid cells, for example voxels (or pixels). These grid cells can then be queried for their centre coordinates and their proximity to the generators to assign each to their closest generator, and to each generator's Voronoi cell its centroid as the mean of its assigned voxels (or pixels). Unfortunately, even though this can be sped up with a proper nearest neighbour search method, this approach is not just inefficient, but it also computes a Voronoi diagram entirely, instead of just its boundaries. This increases storage space and decreases usability for my mesh using application. Luckily, better algorithms have been discovered.

As I said, most of those algorithms are based on obtaining the dual of a Delaunay tetrahedralisation (or triangulation for 2D). As I can implement dual generation myself without inquiring into scientific literature, those methods come down to solving the problem of generating a Delaunay tetrahedralisation for a given set of points. To this end I can again distinguish between two main approaches, known as 'divide and conquer' [7, 8], and 'incremental insertion' [9, 10, 11], although specialised methods exist that fall outside of these categories [12, 13, 14, 15]. For my research, I discuss only the general options.

Divide and conquer is a strategy similar to sweep line for Voronoi tessellations, in the sense that gradually bigger portions of the domain are turned into Delaunay tetrahedralisations until the whole domain is processed. Unlike sweep line, which adds points to the valid tessellation one by one, divide and conquer constructs different valid Delaunay tetrahedralisations and eventually combines these into one whole. The only things needed then are a way to construct the Delaunay tetrahedralisation of some small and simple subset of the points, and a way to effectively combine two subsets into a whole that is still valid. I do not go into explicit solutions here, but refer to the articles referenced.

I cover incremental insertion a bit more extensively, as H. Ledoux [9] convincingly argues that this is the fastest Delaunay paradigm in three dimensions. The idea behind this kind of approach is that the points are added (inserted) one by one, with the Delaunay tetrahedralisation of the points inserted so far staying valid after every insertion; when the last point is inserted, the procedure is finished immediately. In fact this is even closer to the sweep line method for direct Voronoi diagram computation than the divide and conquer paradigm.

Ledoux himself uses a relatively complex three-dimensional variant of the algorithm by Guibas and Stolfi in [6], flipping edges until the empty circumsphere property is satisfied for all tetrahedra, while accounting for all possible degenerate cases. However, the more common Bowyer-Watson algorithm, devised independently by A. Bowyer [10] and D. Watson [11] in 1981, is a bit more straightforward. Their idea is to, instead of flipping edges, delete tetrahedra that break the Delaunay property after an insertion, and refill the resulting polyhedral hole by connecting the inserted point with the hole's boundaries.

When it comes to volumetric meshes existing in three-dimensional space, usually tetrahedral meshes, one of the algorithms mentioned above can be used

to construct the Delaunay tetrahedralisation, from which the dual can be obtained subsequently. The resulting 3D Voronoi diagram can then be intersected with the mesh to obtain the clipped Voronoi diagram, constrained to the mesh, as shown by D.M. Yan et al. [13], which they speed up further by only considering outer (boundary) tetrahedra for intersection [14]. However, this is not exactly what I am looking for. My research interest requires (centroidal) Voronoi diagram computation on triangular meshes; in other words, on 2-manifold surfaces embedded in three-dimensional space. This turns out to be an even more difficult problem, with only few solutions known thus far.

3.1.3 On Meshes

Similar to the clipped Voronoi diagrams I named —three-dimensional Voronoi diagrams constrained to polyhedral meshes— are their polygonal equivalent: restricted Voronoi diagrams, or 'RVD's for short. RVDs are the result of intersecting three-dimensional Voronoi cells with a polygon mesh, and themselves consist of polygonal cells on a mesh surface. An algorithm for their efficient and accurate computation was again presented by D.M. Yan et al. [16]. Despite their complexity in practice, they are one possible candidate for the implementation of my required generate centroidal Voronoi tessellation operator.

Another option, also explored by D.M. Yan et al. a few years earlier [12], is that of conformal parameterisation: mapping the mesh to the plane, sphere, or hyperboloid, constructing the Delaunay triangulation or Voronoi diagram there, and lifting it back to three-dimensional space again afterwards. However, not only does this approach suffer from serious numerical issues when parameterising meshes with complicated geometry, it is also inadequate for meshes of arbitrary topology, because it is, like that of restricted Voronoi diagrams, ultimately a mesh-extrinsic approach, as argued by X. Wang et al. in 2015 [17].

Looking through the existing literature, Wang et al. stated that "to our knowledge, there is no method for computing the CVT on arbitrary surfaces" [17]. To combat this, they came up with their own, intrinsic method. To compute Voronoi diagrams on meshes intrinsically, they use geodesic distances instead of Euclidean distances. Whereas the Euclidean distance between two points on a mesh is defined as the length of the straight line connecting them, their geodesic distance is defined as the length of the shortest (curved) line connecting them that lies completely within the mesh.

They use an exponential map to compute these geodesic distances, and consequently combine them with an incomplete Cholesky factorisation [18] and the fast marching algorithm [19] to find geodesic Voronoi cells and their centroids on the mesh. They can then iteratively turn the found Voronoi diagram into a centroidal Voronoi tessellation. They even compare the CVT convergence speed of their approach plus Lloyd's algorithm with that of theirs plus the superlinear L-BFGS method, but I come back to that in the next subsection. Important here is that the geodesic approach provides a second candidate for the implementation of my required generate CVT operator —combined with some choice of convergence method— and probably an accurater one (Figure 9).

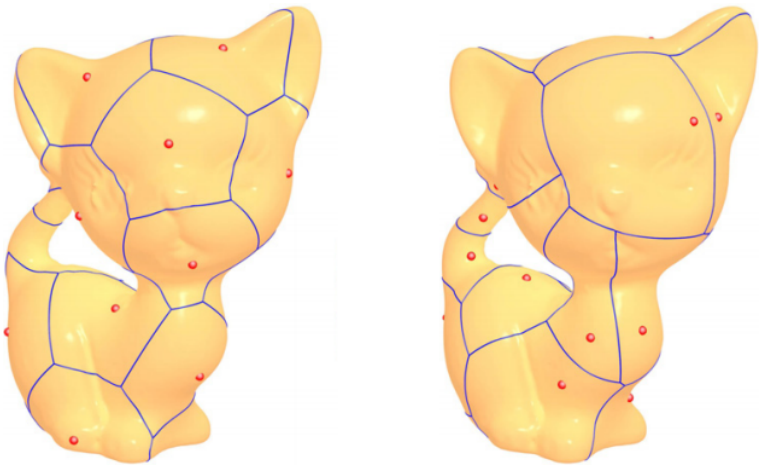

Figure 9: Wang's intrinsic approach for 20 generators on a kitten model (left), and the corresponding extrinsic RVD (right). (reproduced from 'Intrinsic computation of centroidal Voronoi tessellation (CVT) on meshes' [17])

Two problems with this new option, however, are on the one hand its high mathematical complexity, rendering me unable to implement it myself from scratch, and on the other the lack of explicitness in the corresponding article, with no sample code available online. Fortunately, there exists a third (and final) alternative, provided by H. Pan et al. in [1]. Their method focuses on generating constant mean curvature (CMC) surfaces of high mesh quality, including minimal surfaces, which are CMC surfaces with a mean curvature of 0. For this end they, too, utilise centroidal Voronoi tessellations.

Pan's new approach is to minimise a CVT-CMC energy functional in order to obtain a CMC surface of high quality (associated with CVTs). This function involves an extended CVT energy term, to be specific the regular CVT energy with points constrained to the mesh, which is proven to be asymptotically proportional to surface area [1], and a volume term multiplied by a volume weight parameter t. I examine this more closely in this section's concluding subsection on the relation between CVTs and minimal surfaces. For now it suffices to say that sample code of his is available freely for non-commercial purposes, like my project, and seeing as it does more or less what I need (more details to follow in that subsection), I decided to use this for my implementation of the generate centroidal Voronoi tessellation operator, and integrate it within my application.

3.2 Converging to a CVT

So, invented methods for computing CVTs are all iterative ones, and most of them require the computation of a regular Voronoi diagram of a set of points during every iteration. This even holds true when computing them on meshes, as I need be able to. Back to the main categories I distinguished in this section's

introduction, there are those approaches which go from Voronoi diagram to Voronoi diagram using Lloyd's method [1, 12, 17, 20] until this process converges to a centroidal Voronoi tessellation, and those that attempt to minimise a CVT energy functional, defined as the sum of all Voronoi cells' integrals of their points' squared (Euclidean/geodesic) distance to their generators, and can converge superlinearly in order to obtain the new generator positions corresponding to that particular CVT configuration which is closest to their original locations [1, 16, 17, 20, 21, 22]. But which kind of approach would be better to use?

Rate of convergence, whether linear or superlinear or even quadratic, is understood as the speed with which an iterative progress gets from its initial configuration to a stable, unchanging limit. Linear convergence means that every iteration brings the configuration for approximately an equal amount closer towards its limit, while superlinear convergence means that each iteration has a bigger impact than the previous one. In general, superlinear methods are faster than linear methods, increasingly so for larger meshes with a greater initial distance from a solution (CVT), which require increasingly many iterations.

This seems to indicate that I better use a superlinear convergence method, but next to being more difficult to implement, these are also less robust. Therefore I test both options, and compare the results. But before I can do that, I give an overview of the convergence algorithms provided in the scientific literature on computional geometry. Starting with a more detailed explanation of Lloyd's method, I describe multiple superlinear methods next —primarily the widely used L-BFGS method— and finally discuss multigrid approaches shortly, which are ways of further speeding up the other methods presented.

3.2.1 Lloyd's Method

In 1982, S.P. Lloyd invented his algorithm, which was named after him [2]. The most widely used algorithm for turning Voronoi diagrams into centroidal Voronoi tessellations since the beginning of CVT research, it is also the most straightforward one. As soon as one is able to compute a regular Voronoi diagram of a set of generating points in the required number of dimensions, for example with one of the approaches mentioned in the previous subsection, the remaining steps become simple: just compute the Voronoi cells' centroids, move their generators there, and repeat the process. Stop when the total or average amount of movement has dropped below a prespecified convergence threshold, and the CVT approximation is done. I refer to Section 2, on concepts, where I already discussed Lloyd's method in one sentence, for an illustration (6).

3.2.2 Superlinear Methods

Recently, Lloyd's method has gotten a lot of competition from quicker converging methods, now that the research field is maturing and people start to realise that not just the accuracy, but also the speed of a CVT generation algorithm is vital to its success [20]. The breakthrough was the characterisation of the centroidal Voronoi tessellation generation case as an energy functional minimi-

sation problem, and later the proof by Y. Liu et al. that this function has C^2 smoothness, meaning that its first and second derivatives are continuous, and are not just of C^0 parametric continuity as previously thought [23]. The latter observation made it possible for researchers to draw from an array of mathematical optimisation tools to speed up the old-fashioned linear convergence attained by Lloyd, including Newton's method [20], quasi-Newton methods [1, 16, 17, 21, 22], conjugate gradient methods [21], and gradient descent methods.

The most promising of these improved methods, at least in terms of the extent to which it has been adopted, is the quasi-Newton L-BFGS (Limited-memory Broyden-Fletcher-Goldfarb-Shanno) method [24]. A memory-efficient variant of the original BFGS algorithm, it estimates the CVT energy functional's inverse Hessian matrix by evaluating its gradient, and uses it to search variable space more effectively. Even though J.C. Hateley, H. Wei, and L. Chen found their conjugate gradient implementation to be a slightly faster than L-BFGS [21], it is nonetheless the latter that is becoming most widely used, helped by the fact that multiple L-BFGS implementations are freely available.

Hao Pan et al., whose code I integrated into my own application as detailed by the next section, utilise the HLBFGS (Hybrid L-BFGS) library by Y. Liu, which unifies the L-BFGS method with preconditioned L-BFGS and preconditioned conjugate gradient methods, similar to those by Hateley, Wei, and Chen, and can also function like a gradient descent or as Newton's method [1, 21].

I do not delve deeper into these superlinear methods here, as this is an extensive, highly mathematical research field on its own, and apart from the above outside of the scope of my thesis. I stick to Lloyd's method and L-BFGS because of their good, well-known qualities, and because they are also the methods used by Hao Pan's application [1]. I further discuss his code and the corresponding scientific article as a bridge to the next section on implementation and integration, but first I provide a short summary of multigrid approaches.

3.2.3 Multigrid Approaches

The idea of a multigrid approach is to first solve a problem, in this case that of constructing a centroidal Voronoi tessellation, on a coarse grid, before converting the coarse grid to the actual fine grid, where the problem is solved again [20, 21]. Because the problem can be solved much faster on the coarse grid, and its solution provides an initial configuration on the fine grid that is a lot closer to an actual solution than the input itself, the total computation time can be reduced. For example, Q. Du and M. Emelianenko construct a CVT by applying Lloyd iteration until the solution is found to be reasonable close, and then switch to Newton's method to finish the job [20]. For this to work well, the amount of generators in the input should be sufficiently large, with their number greatly reduced on the coarse grid, and a proper refinement method for conversion to the fine grid should be supplied. For example, J.C. Hateley, H. Wei, and L. Chen demonstrate a robust refining method that multiplies the amount of generators by four, while preserving the relationship between the final Voronoi regions that resulted from the course grid iterations [21]. Figure 10 below illustrates it.

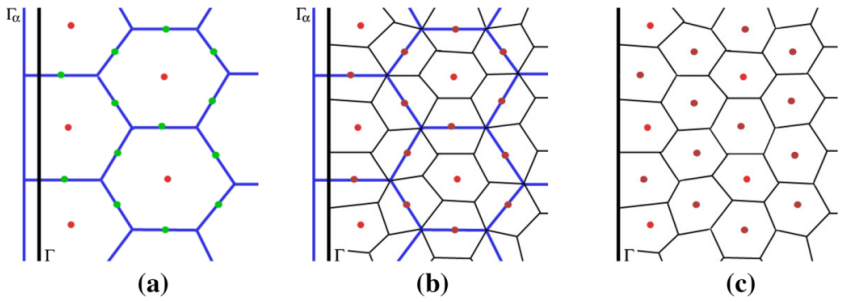

Figure 10: Hateley's, Wei's, and Chen's refinement method. (reproduced from 'Fast Methods for Computing Centroidal Voronoi Tessellations' [21])

Multigrid approaches can also entail three grids, or even more, of varying density. The optimal amount of grids scales with the problem size. Usually though, two grids are enough for moderately sized problems, such as in the articles referenced above. Approaches like those are sometimes referred to as two-grid methods. Myself I do not use any such method however, as Hao Pan's application does not do so either, but it probably could speed up his code and consequently my application, certainly when the amount of grids is based on mesh size automatically. This concerns possible future work (see Section 6).

3.3 Relation to Minimal Surfaces

Hao Pan and his colleagues [1] were aware that constant mean curvature surfaces can be approached by minimising surface area subject to some volume constraint. However, they were dissatisfied with the final triangle quality of existing approaches for converging meshes to a CMC surface this way, and looked for a way to improve on this. After proving that CVT energy is asymptotically proportional to squared surface area, they were able to swap out the area term for a CVT term, and thereby obtained their novel CVT-CMC functional. This function is defined as extended CVT energy —regular CVT energy but with integrating only over points that lie directly on the mesh surface– plus mesh volume multiplied by the volume weight parameter t (Figure 11).

Minimising this energy functional produces good approximations of CMC surfaces, while keeping the mesh quality high during the whole convergence process. This process can be seen as balancing the accurate computation of a centroidal Voronoi tessellation with some volume preservation. However, in order to obtain good results, boundary vertices remain fixed at all times. When the boundary loops of the initial open mesh correspond to the boundaries of a known minimal surface, while using $t = 0$, the result very closely resembles that minimal surface. The fact that ignoring the volume term by setting t to 0, thereby disregarding volume preservation entirely, results in a minimal surface, shows that (extended) CVT energy is indeed related to minimal surfaces; or in other words, to smooth surfaces that minimise area between fixed boundaries.

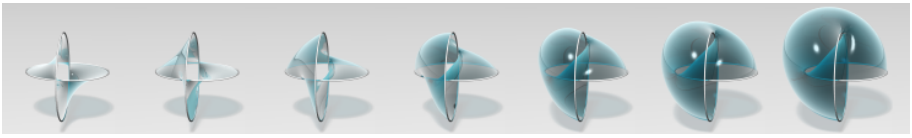

Figure 11: Pan's results with boundaries of two interlaced rings, while varying the volume weight t. The mesh on the left is a minimal surface with $t = 0$. (reproduced from 'Robust Modeling of Constant Mean Curvature Surfaces' [1])

Hao's code corresponding to the article in question [1] is written in `C++` and is freely available for non-commercial purposes. This allows me to integrate it within my own application, as detailed by the next section. It supplies convergence using either Lloyd's algorithm or the superlinear L-BFGS method, and because I could not clearly tell beforehand which is better for my personal research purposes, I test both of them in Section 5 on experimental results.

Although this approach provides me with a good implementation of the generate CVT operator on polygon meshes embedded in three-dimensional space, as I searched for in this section, it is worthwhile to note that it does not do exactly what I had wanted. As H. Pan et al. use simple Euclidean distances, and not geodesic distances such as those used by Wang et al. [17], vertices are actually allowed to move away from the mesh surface during an iteration. As the centroid of a Voronoi cell on a convex mesh lies just within the mesh, the CVT limit is not computed on the input mesh, but is instead converged to by iteratively shrinking the mesh. This poses no problem for me though, because boundary vertices stay fixed, and the required smoothing effect remains. Moreover, this even brings generate centroidal Voronoi tessellation more in line with my other smoothing operators, which all entail a shrinking effect too.

4 Implementation

As I mentioned, I started from an application originally intended for loading meshes to display them, perform Loop subdivision on them, and again display the results. Next to different display modes (point cloud/wireframe/solid) and shading options (smooth/flat), it allowed for rotating, scaling, and translating the mesh view. Written in `Qt` and `C++`, with `OpenGL` used for mesh visualisation, I modified and extended this application with code of my own to comply with my own research interest. This involved adding three operators, and more.

To give a proper overview of my approach, I first describe the data structure used by the application to store meshes, known as the half-edge data structure, or as the doubly connected edge list. Then I explain the `Wavefront OBJ` geometry definition file format, which is the input format importable by the application, and to which it can now also export again. Finally, I detail more extensively my personal technical contribution, including code modifications, added features, and the resulting overhaul of the application's user interface.

4.1 The Half-Edge Data Structure

The half-edge data structure is the data structure used by my application to internally represent manifold meshes. Even though I work with triangular meshes, it can represent other polygon meshes as well, which makes implementing the generate dual-dual operator possible. This data structure is often used for the implementation of computational geometry algorithms, and in particular those subdivision related. Although relatively expensive to maintain, its high and cheap usability makes it a convenient choice.

A 'half-edge' is exactly what the name implies: half an edge. Every mesh edge between two vertices is stored as a set of 'twins', to be specific one half-edge pointing to the first vertex and one half-edge pointing to the second. The nice thing about this is that two twin half-edges, which together constitute one edge, are part of two different faces, which are adjacent because they share a common edge. Hence face adjacency information can be extracted directly from half-edges' twin information, as can also be seen from Figure 12 below.

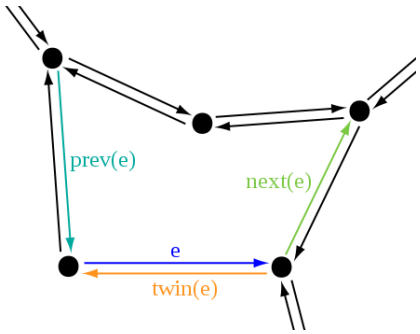

Figure 12: The half-edge data structure illustrated. (courtesy of Wikimedia)

Next to its twin, each half-edge stores a pointer to its incident face, and to its next and previous half-edge within that face (12). They also store a pointer to their target vertex, with each vertex storing a reference to one outgoing half-edge plus its own coordinates. Faces only need to store a reference to one of their sides (half-edges), but I have them precompute their valency too, as this speeds up their processing. Everything together, this structure allows for fast traversal of the represented mesh, while requiring relatively little storage space.

4.2 Wavefront OBJ Files

One of the most straightforward and widely supported geometry definition file formats used for representing meshes is `Wavefront OBJ`. Files in this format take the `.obj` extension, and their simple variants are used as input, and possibly output, of my application. With 'simple variants' I mean those `OBJ` files representing triangular meshes, that define vertices and faces only, and thus contain no comments, vertex normal definitions, texture coordinates, et cetera.

The input format accepted comes down to a list of vertex definitions followed by a list of face definitions, with one line of text per definition. A vertex is defined as '`v [x-coordinate] [y-coordinate] [z-coordinate]`', whereas a face is defined as '`f [vertex-index] [vertex-index] [vertex-index]`'. The index of a vertex is the line number of its definition, with counting starting from one accordingly. A `Wavefront OBJ` file ends with an empty line.

When starting my application it asks for an `.obj` file to be loaded. A variety of those files already come with it, but users can supply their own as well, and any can be chosen to then apply the available operators to. Before the mesh is displayed however, the file is read and converted to the half-edge data structure. All operators work on the latter representation, and only when the modified mesh is in need of exportation, is it converted back to the original file format.

4.3 Initial Modifications

Before I get to a more detailed description of what features I added and how, majorly the three extra smoothing operators, I first shortly summarise the modifications I made to the code base that I started out with. Most importantly, as the original application was focussed on performing Loop subdivision, I had to decouple this algorithm so that it could be treated as just a single operator, and other operators could be added to the mix. I also switched the default (initial) display and shading options to wireframe and smooth, respectively, for optimal insight into the effects of the different operators on the input mesh.

Furthermore, as the `OpenGL` part was written to purely support triangular meshes, I adapted it so that it could handle arbitrary polygon meshes, and could therefore also correctly display results of the generate dual operator. Even though this operator did not make it to the final application, in favour of the generate dual-dual operator, this can still be useful for those who do want to use my application to experiment with an odd amount of dual mesh computations, and the like. Lastly, I made a lot of tiny modifications to improve the code's

legibility, speed, and extensibility, but I only discuss those that concern the application's user interface, in the last subsection of this section.

4.4 Added Features

First however, I describe implementation details of all added features. I added 'undo' and 'redo' buttons to quickly switch between computed meshes, so that applied operations can be undone and redone without having to perform the calculations again. This also means that the whole sequence of meshes, starting with the initial (input) mesh and then one modified version of it for every operation applied, is kept in memory while the application is running. Still, this does not require much memory, and meshes which become unreachable due to undoing them and then applying another operator are removed from memory immediately. This sequence of meshes is referred to internally as the 'model'.

Now to discuss the addition of the three new smoothing operators; Laplacian smoothing was not difficult to add. What I did was copy the latest mesh from the model, in other words the one currently displayed by the mesh view of the application, and then change coordinates for each vertex by traversing its neighbours, calculating their average position, and then relocating the vertex towards it. Here the smoothing weight parameter w was taken into account, defaulting to the safe value of 0.5 as stated, which means that all vertices are only moved halfway towards the average position of their neighbours.

Implementing generate dual, and consequently generate dual-dual, was a bit more complex. Although the concept of this operator is relatively easy in theory, the half-edge data structure renders it moderately difficult in practice. Using this structure, it is straightforward to find the centroids of the faces of the current mesh, construct new vertices there, and connect them by new half-edges based on current face adjacency information. However, filling in the 'twin', and even more so the 'next' and 'prev' fields of new half-edges is not. I managed to solve this by replacing every current half-edge by one that points to the new vertex associated with its incident face. The twin of that new half-edge is then found as the new half-edge corresponding to the current half-edge's twin. Afterwards, a new half-edge's next is derived as the twin of the previous of the corresponding current half-edge, and then the new half-edge corresponding to that one. This way, new previouses and new faces are easily identified too.

This idea might be difficult to grasp, but for the purposes of this thesis it suffices to understand that I did successfully implement the generate dual-dual operator, before switching my research focus to open meshes and accordingly deciding that I would not be able to use it anymore.

As the third and last new smoothing operator, I added generate centroidal Voronoi tessellation. As stated in the previous section, I did not implement this myself due to its high mathematical complexity, but was able to integrate code by Hao Pan, from the University of Hong Kong, directly into my own application [1]. This was not easy either, but I managed to connect his code to mine seamlessly. To this end, I examined his application's inner workings extensively, added the source files to my own project, removed redundant classes, changed

some internal behaviour, changed my own code to communicate with Hao's classes, and made sure everything compiled correctly together. To communicate, my main class exports the currently displayed mesh to `OBJ` format, feeds this to Hao's code, and imports the resulting `OBJ` file again when it has finished.

Finally, I contributed two more substantial methods to the code, which I will now name without going into further details regarding their implementation, as they are ultimately of minor importance compared to the others. First off, exporting the modified mesh back to an `OBJ` file. This is required for generating centroidal Voronoi tessellations, as clarified above, and is a handy option to have apart from that as well. Secondly, one for triangular mesh volume calculation. This allows for checking the change in mesh volume after applying an operator, which gives some information about its proximity to convergence, especially for those operators other than generate CVT, which computes the resulting mesh's CVT-CMC energy for similar knowledge. All added methods contain comprehensive comments so that interested researchers can easily understand them, and continue the application's development for their own pursuits.

4.5 Updated User Interface

Now that I modified the initial application, and added features to it, it was time to give the user interface an overhaul. Not only to reflect the new possibilities and my personal research goals, but also because it was not suited to small laptop screens originally. The user interface existed of a mesh view at the right side, and at the left a sidebar with viewing options and a button for the Loop subdivision operator. I sticked to this lay-out, removing unnecessary sidebar elements, moving up old sidebar elements, and adding new elements to it.

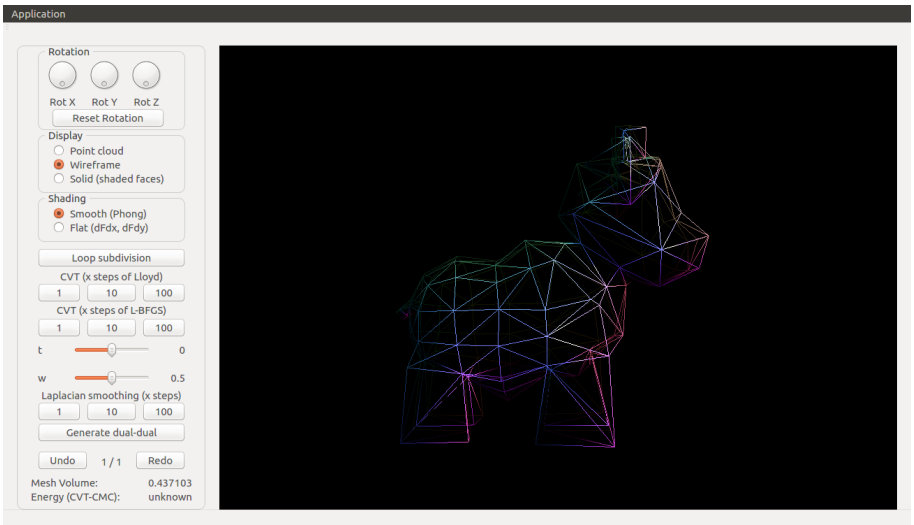

Figure 13: My application's final user interface, displaying the Spot mesh.

As can be seen from Figure 13 above, I added buttons for the other three operators: Laplacian smoothing, generate dual-dual, and generate centroidal Voronoi tessellation, with the latter receiving options for Lloyd's algorithm as well as for the L-BFGS method. To simplify my research, which results I discuss in the upcoming section, I included the functionality for performing either 1, 10, or 100 steps at once for both Laplacian smoothing and the generate CVT variants. Increasing convenience further, I integrated horizontal sliders to facilitate changing the CVT-CMC volume weight parameter t and the Laplacian smoothing weight w from within the user interface, restricted to sensible values.

Lastly, I updated the sidebar with some extra information, to provide a better overview of what is going on. Between the undo and redo buttons, a mesh counter is visible, which shows both the current mesh number within the model and the total model size itself. Below that, as the bottommost sidebar elements, are listed the current mesh's volume and, if the last operation was a CVT one, its CVT-CMC energy, to provide important information on convergence.

5 Experimental Results

Can the implemented operators now together converge to a smooth version of the input mesh, and if so, in what order to obtain the smoothest appearing mesh for the lowest computational cost? To find an answer to this two-sided question, I first explore the effects of the four smoothing operators separately, before looking into more complex cases, for which I combine them in various orders. So far as possible I focus on (various) open meshes, as argued for in the previous sections. Although images of resulting meshes form my most important results, I back them up with amount of iterations until convergence, final volume and energy, and average volume and energy reduction per iteration.

Note that open meshes can sometimes still be assigned a volume by considering their convex hull. However, as I am not interested in exact volumes but rather in information on convergence, I simply use a naive method for computing the volume of closed triangular meshes, as this is more efficient. It holds true anyway that when this value no longer changes, the mesh has converged.

5.1 Effects of Single Operators

Evaluating the effects of single smoothing operators on an input mesh allows me to discover their own specific properties, and to find their convergence limits. For generate CVT, both the Lloyd and the L-BFGS variant are said to have converged when the CVT-CMC energy stabilises, while for generate dual-dual and Laplacian smoothing this is instead found as stabilised mesh volume. Loop subdivision, whose role is adding vertices, as opposed to other operators' redistribution of vertices, never converges in practice. After infinitely many iterations it converges to a smooth limit surface, but I cannot reach this. As stated in Section 2, it just keeps adding vertices, until my laptop can no longer handle the amount of mesh data. For most, relatively small input meshes considered, my laptop still runs smoothly after 3 Loop subdivision steps, becoming slow after 4 steps, and very slow after 5. For this reason I do not experiment with more than 3 steps for the complex cases investigated later, and go up to 5 here.

5.1.1 Loop Subdivision

In this subsection, my open test mesh of choice is one depicting a Moai statue, which consists of 891 vertices and 1721 faces. Initially, its volume is 1.16016. After one step of Loop subdivision, the volume becomes 1.15491. After two, three, four, and finally five steps, it drops down further to 1.15362, 1.15331, 1.15326, and 1.15262, respectively. At the end of this procedure, the Moai mesh has 882128 vertices and 1762304 faces. This density is way too high for my research purposes, but this goes to show how quickly Loop subdivision increases mesh density. The slowly and irregularly, but surely decreasing mesh volume and Figure 14 below indicate that the Moai indeed becomes smoother.

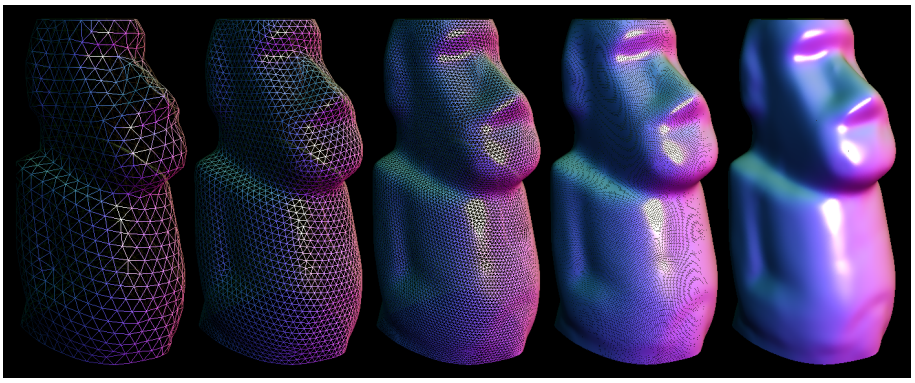

Figure 14: The Moai mesh after 0, 1, 2, 3, and 4 steps of Loop subdivision, respectively. For 5 steps the result looks the same as after 4.

5.1.2 Laplacian Smoothing

Using the same initial Moai mesh, I test Laplacian smoothing with fixed boundary vertices. Again, at the start the mesh's volume is a mere 1.16016. When the Moai converges after about 4600 Laplacian smoothing steps, its volume has dropped to -0.120711. Note that this value is negative because I use a straightforward volume computation algorithm originally meant for closed meshes. Halfway until convergence, so after 2300 steps, the volume was already down to -0.120348. This shows that Laplacian smoothing requires many iterations to converge, with volume reduction per iteration decreasing quickly. The latter result is also observable from the corresponding figure (15) below, which shows the input mesh contracting to a thin model; first quickly, then slower.

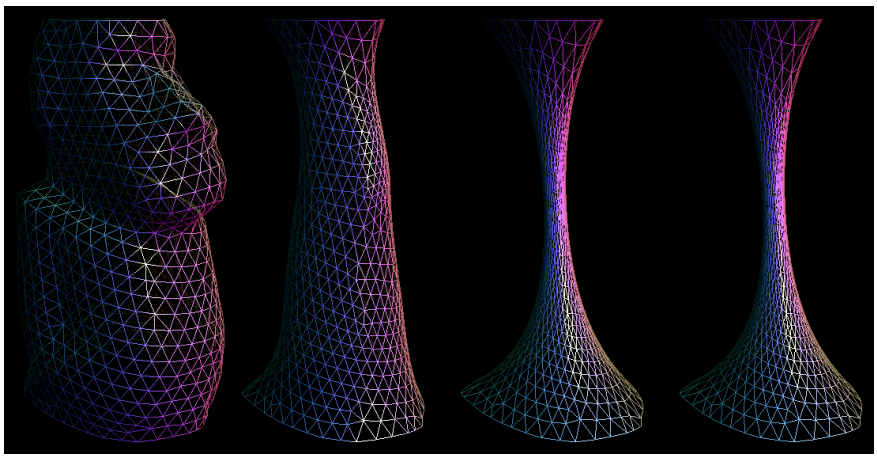

Figure 15: The Moai mesh after 0, 100, 2300, and 4600 steps of Laplacian smoothing, respectively.

This experimental result was obtained using the default value of 0.5 for the Laplacian smoothing weight parameter w. When using the less safe value of $w = 1$, the smoothing becomes stronger, and the mesh converges after about 2600 Laplacian smoothing steps with a volume of -0.120714. Thus, the final result is slightly better, although in fact as good as the same, and almost twice as fast. Hence, increasing w is in most (general) cases a smart thing to do. One should keep in mind however that in my experiments values of w above 1.3 provided chaotic non-converging results, and as this is always a danger when using a value above 1 for any arbitrary mesh, I do not recommend doing so.

5.1.3 Generate Dual-Dual

To test the properties and possible convergence limit of my generate dual-dual operator, I apply it to a closed mesh: a model representing a cow, named Spot. Shown in Figure 16 below is Spot as she gets smaller and smaller from repeated application of dual-dual generation. As expected, her asymptotic limit is the relocation of all initial vertices to one and the same position. Although I cannot practically reach this limit, it suffices to say that Spot became invisible after 302 steps, with at that point a volume of $4.17873e-13$. Next to this it is worthwhile to point out that also here volume reduction decreases every iteration, and that this operator performs very similar to Laplacian smoothing with $w = 0.7$.

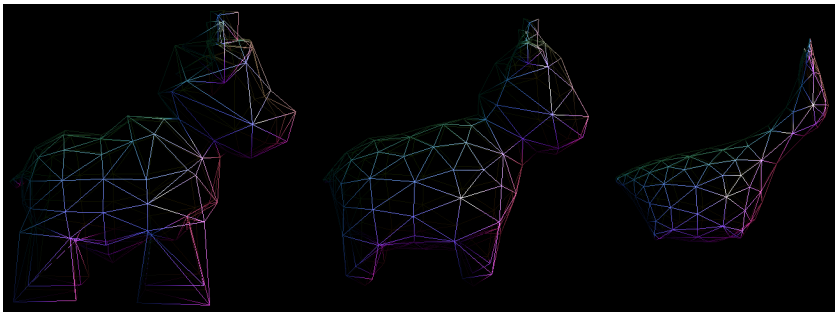

Figure 16: The Spot mesh after 0, 1, and 10 generate dual-dual steps, respectively.

5.1.4 Generate CVT

Back to the open Moai mesh, I find the convergence limits of the Lloyd and the L-BFGS variant of the generate CVT operator, as well as their other interesting properties. As it turns out, Lloyd's algorithm converges after 769 steps, with a final CVT-CMC energy value of 0.000590755 and a volume of -0.400906. L-BFGS, on the other hand, converges after as little as 101 steps, with an energy of 0.00116516 and a volume of -0.401582. Although Lloyd's algorithm manages to minimise energy somewhat further, and hence better approximates a centroidal Voronoi tessellation, L-BFGS is much faster, despite one L-BFGS step taking a

little longer than a Lloyd one. Moreover, although negligible, the latter approach actually results in a slightly lower volume, indicating a smoother result despite lower triangle quality. Figure 17 below shows a problem with L-BFGS however, in the form of artefacts not observed when converging using Lloyd's algorithm.

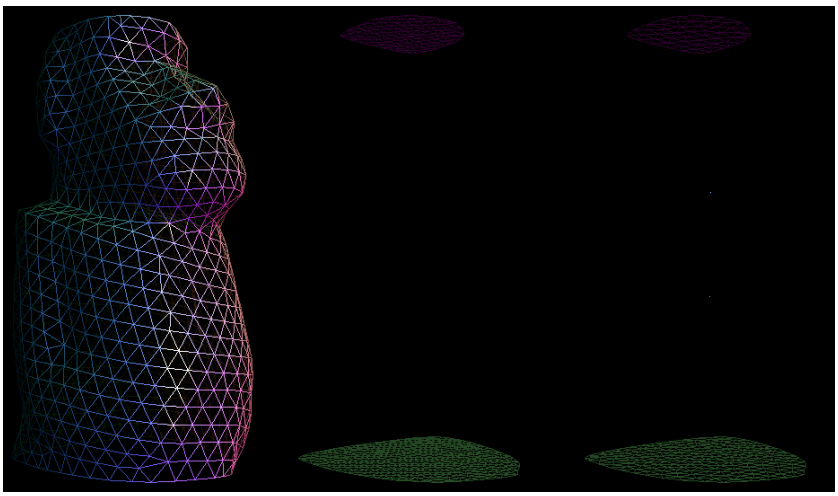

Figure 17: The Moai mesh: input (left), converged using Lloyd's algorithm (middle), and converged using L-BFGS (right, containing artefacts).

Looking closely, one can see tiny spheres floating between the boundary planes. The problem is that L-BFGS avoids geometrical computations, and instead just mathematically searches for a minimum of the CVT energy functional. Despite being a lot faster, this has some downsides. First off, the mesh no longer morphs 'logically' from the initial mesh to its stable smooth state, but appears strange to the human observer until the minimum is reached. And secondly, following from this, it often produces artefacts as those shown. Even more striking than this though, is the fact that both variants of the generate centroidal Voronoi tessellation operator result in a disconnected mesh here.

As expected, boundaries are kept fixed when applying this operator, but while trying to find a smooth surface of high quality between the boundary loops, generate CVT finds the optimal solution to consist of two separated surfaces. Although there is no denying that the two resulting boundary planes are smooth and of high quality, and even though this effect makes sense retrospectively, this is clearly not what I want. What I want is to automatically obtain a smoothed version of the input mesh, in such a way that this would satisfy a user's needs and expectations. For that to happen, the final mesh should still resemble the initial one, and hence it should never split up during the convergence process.

Fortunately, I can use the CVT-CMC energy functional's volume weight parameter t to my advantage here. In the example above, this parameter was set to 0, meaning that the methods approached a centroidal Voronoi tessellation without balancing this with some volume preservation. By using a negative

value for t instead, volume is to some extent maximised when the function is minimised, and therefore preserved. After testing different values for t while judging quality from my own perception, I obtained the optimal results for the Moai mesh when using $t = -0.002$. Increasing this value in my application still gives disconnected limits, while decreasing it further makes the results more spherical and less resembling the input. Figure 18 below shows these results.

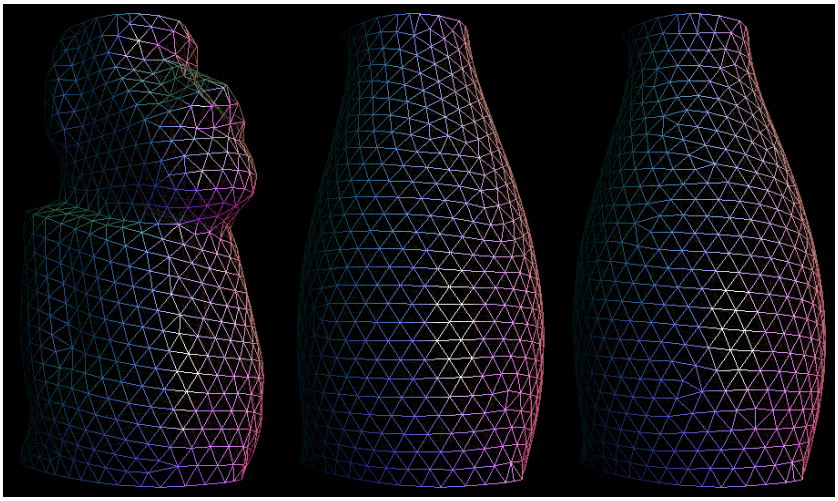

Figure 18: The Moai mesh with $t = -0.002$: input (left), converged using Lloyd's algorithm (middle), and converged using L-BFGS (right).

This time around, Lloyd converges after 1394 iterations with an energy of 0.00115628 and a volume of 1.58888, with L-BFGS converging after 178 iterations and a final CVT-CMC energy of 0.00113129 and a mesh volume of 1.65923. The conclusion here is that finding the optimal value for the volume weight parameter before applying generate CVT allows me to obtain a smooth mesh of high quality just as well, but now with the resulting mesh remaining one whole that still resembles the input mesh a bit, at the cost of almost double the computation time. But if this is required for acceptable results, so be it.

5.2 Complex Cases

Now that I have investigated the effects of single operators on meshes, and found their convergence limits, I can devise a proper strategy for tackling more complex cases, to which I can apply my full arsenal of operators. Well, almost; as I focus on open meshes for my research, as explained before repeatedly, generate dual-dual is not applicable. I consider two cases here that together provide for all remaining conclusions necessary to allow me to ultimately answer my main research question in a nuanced and complete way, without supplying redundant information. The first one consists of open cylinder meshes of various heights.

The second utilises boundary loops of different shapes in an attempt to nicely morph between them using my three still available smoothing operators.

I stress the importance of convergence here. Convergence means that reapplying the latest applied operator will no longer change the mesh. This is the property of my work that allows the smoothing process to automatically stop when the input mesh has become smooth enough, removing the need for manual interference. However, as is clear from the images in the previous subsection, Laplacian smoothing and generate CVT have different convergence limits, while Loop subdivision has none of practical use. As I want the input meshes to converge, I therefore have to end with either repeated application of Laplacian smoothing or with repeated application of generate CVT. My choice goes to the latter option, because when using the volume weight t I can clearly converge to nicer results this way. This leaves Laplacian smoothing and Loop subdivision as initial operators to alternate between, before making the switch to centroidal Voronoi tessellations. I experiment with different alternations and switching moments to see what order provides the best results in the shortest time.

5.2.1 Varying Distance between Boundary Loops

As the first of my two more complex case studies, I consider three open cylindrical meshes with the same radius, but with different heights. Similar to the Moai mesh used for tests so far, these contain two circular boundary loops in parallel planes. Unlike the Moai's, the cylinders' two boundaries are perfect circles, and are of the exact same radius while being precisely parallel. This allows for more controlled experiments, and provides for additional interesting conclusions.

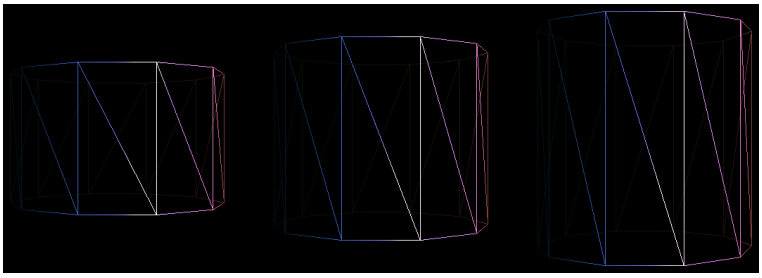

Figure 19: The three cylinder meshes of different height.

Above (Figure 19) are aligned the three cylinders, each having a radius of 1, with their height, from left to right, being 0.6, 0.8, and 1.0, respectively. Before I can converge using the generate CVT operator, I have to use at least one Loop subdivision step, to make sure there exist non-boundary vertices which can be redistributed. When achieving convergence afterwards, an interesting phenomenon can be observed, as illustrated by the image below: the two smaller cylinders each converge to a stable catenoid, as is expected from minimal surface theory, related to CVTs, but the third converges to two separate boundary planes, as could be expected from my research in the previous subsection.

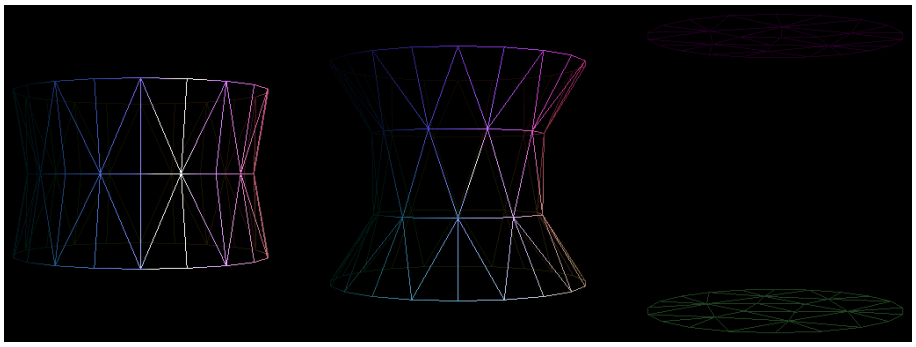

Figure 20: The three cylinder meshes of different height, after convergence.

Figure 20 above proves that the distance between boundary loops is critical: when the boundaries are close enough, the volume weight parameter does not have to be taken into account to achieve a good result, and it can be left at 0. However, as I found when examining my second complex case study, this only holds true in general when the boundary loops are of similar shape. When these shapes are different —which was to a lesser extent already the case for the Moai mesh— using t is unavoidable if one wants to converge to a smooth mesh which still sufficiently resembles the input mesh, at least when considering just generate centroidal Voronoi tessellation combined with my other operators.

Now that I know when polygon meshes become disconnected, why they do so, and how to combat this to moderate success, I can focus on the other result obtained here: the first two cylinders converging to stable catenoids. I want to find out in what order I can best apply Loop subdivision and Laplacian smoothing, and how many times, before switching to generate CVT until convergence, for such a kind of input mesh for which I know that it has a good limit.

I test the different possibilities, keeping track of final cylinder volume and CVT-CMC energy versus amount of iterations, to find an answer to this important subquestion. To this end, I use the first cylinder, the one with a height of 0.6, to perform experiments on. Note that I stick to Lloyd's algorithm for CVT convergence for the remainder of the thesis, as this is the most robust choice, and I have already discussed how it compares to L-BFGS.

Again, I have to start with at least one step of Loop subdivision, to make sure there actually exist non-boundary vertices to redistribute, in this case for the Laplacian smoothing operator. My logical remaining options then are using Loop subdivision only and then switching to Laplacian smoothing, or alternating between them. I experiment with both options up to 3 Loop subdivision steps, to make sure my laptop keeps running smoothly as mentioned before. For a small mesh like this cylinder, I decide to use Laplacian smoothing batches of 10 steps, to make sure the impact is comparable to one step of Loop subdivision. I provide the results of these tests in tabular format (see the table below).

alternation?	#Loop steps	#CVT iterations	volume	energy
no	1	12	0.235909	0.0160822
no	2	144	0.128302	0.00280128
no	3	558	0.131937	0.000676026
yes	1	12	0.235909	0.0160822
yes	2	161	0.1331	0.00282004
yes	3	450	0.141936	0.000679621

Note that there is no difference between the first and the fourth row, because they both correspond to performing one step of Loop subdivision followed by ten steps of Laplacian smoothing, before converging using Lloyd. More importantly, two new conclusions can be made. First off, adding more vertices by means of Loop subdivision greatly increases the amount of required iterations and hence the required computation time, but at the same time greatly reduces final CVT-CMC energy. There is no clear optimal approach: it is up to the user to balance between quality of the final smooth mesh and the time required to compute it. Secondly, although the differences are very small, alternating Loop subdivision and Laplacian smoothing on average reduces total computation time, at the cost of higher resulting mesh volume and energy. In other words, the two conclusions here are similar: when it comes to automatic mesh smoothing, the optimal order of smoothing operator depends on the user's specific demands, balancing speed versus quality, except that one should always end with generating CVTs.

5.2.2 Morphing from One Letter to Another

When changing the shapes of the two parallel boundary loops to be different from each other, and after initially connecting these boundaries with simple straight lines, I can use my operators to smooth the created surface. This approach allows me to obtain a nice morph from one shape to another. As an example, I create a surface between an open 'S' shape and a similar 'T' one, and turn it into a smooth morph between the two letters by first alternating between Loop subdivision and Laplacian smoothing, and then converging with centroidal Voronoi tessellations. Below is depicted the input (Figure 21).

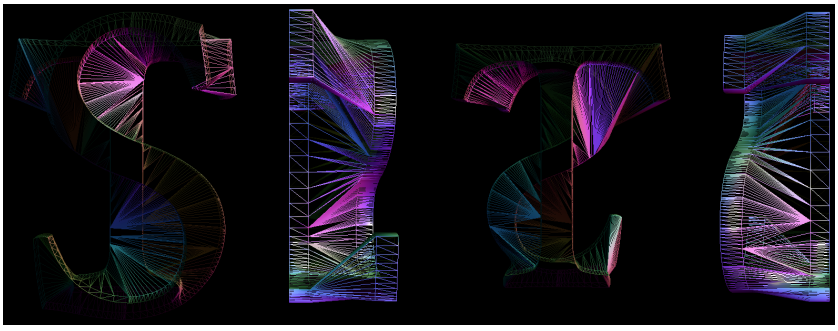

Figure 21: The initial S-T mesh, viewed from its front, left, back, and right.

Now what I would expect from my cylinder experiments is that by initially placing the 'S' and 'T' very close together, the mesh would converge to one smooth whole without having to touch the volume weight parameter. However, this does not happen. Apparently, the generate CVT operator only converges to good results for $t = 0$ when the boundary loops are of similar shape. Once more, despite providing less-than-optimal results, I have to find its optimal value for this particular mesh in order to obtain at least some proper result. Again, I pick the largest (least negative) value for t for which the mesh surface does not separate into two parts, and find this to be -0.005. Now the mesh does stay connected, but as I had feared, it has become too spherical (Figure 22).

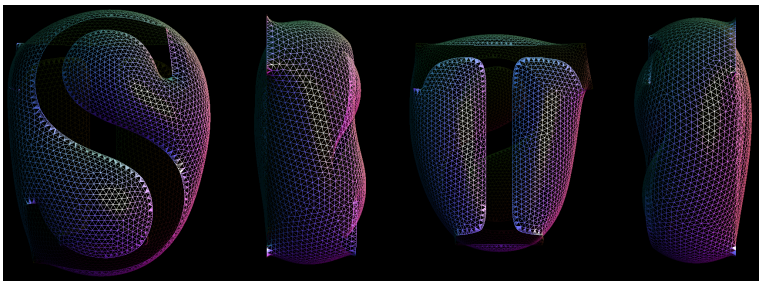

Figure 22: A converged S-T mesh, viewed from its front, left, back, and right.

Exact energy, volume, and iteration metrics do not matter anymore here, as I have gathered enough conclusions by now to draw from a nuanced and complete answer to my main research question. One important peculiarity provides extra depth to this answer however. This is the fact than I can actually obtain a smoother appearing morph than the one shown above (22), within less iterations. This result is illustrated in Figure 23 below. It was achieved by applying just one Loop subdivision step and a hundred of Lloyd's method. The only problem: in no way has the mesh converged at this point. But in an optimal scenario, I should be able to stop here. This requires a new convergence threshold, that generate CVT and its colleagues cannot provide on their own. This forms the primary future work on this subject, as detailed by the next and final section.

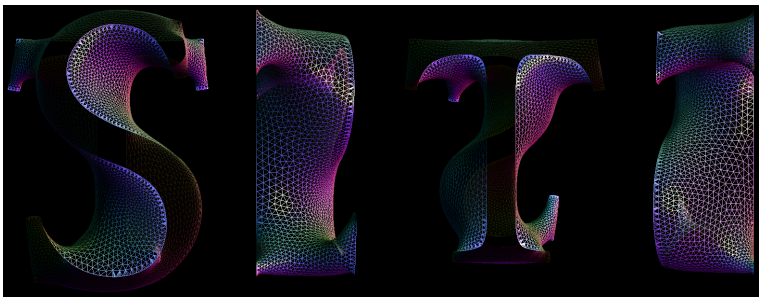

Figure 23: A smoother S-T mesh, viewed from its front, left, back, and right.

6 Conclusion

Recall my research question: "Given an initial mesh, can the operators of Loop subdivision, Laplacian smoothing, generate dual-dual, and generate centroidal Voronoi tessellation together converge to a smooth variant of high quality when boundaries are kept fixed, and if so, in what order to achieve the best result?".

In fact, as became clear during the thesis, this question contained multiple subquestions. First off, I wanted to find out how feasible it is to apply the CVT concept to the smoothing domain, combined with more traditional smoothing algorithms. Secondly, I wanted to obtain smooth results still resembling the input triangular mesh automatically, so by means of convergence and not by means of manual interference. And lastly, I wanted to do this as quickly as possible, and without introducing too many new vertices to keep computational overhead low, with the results still appearing acceptable to the human observer.

Along the way, I decided to focus my research on open meshes instead of on closed meshes, allowing me to keep boundary vertices fixed conveniently, but forcing me to discard the generate dual-dual operator. Nonetheless, this allowed me to carry out all the necessary experimental tests, after introducing important concepts, carrying out literature research, and describing the implementation details of my application which I used for performing these experiments. I drew various conclusions, and they all contribute to a nuanced and complete answer to the research question, which I can now fully provide.

Loop subdivision, Laplacian smoothing, and generate centroidal Voronoi tessellation can in some cases indeed converge together to a smooth mesh of high quality, when the input mesh is an open mesh whose boundary vertices remain fixed. To achieve this, first apply Loop subdivision and Laplacian smoothing, alternating for speed or sequentially for quality, and apply generate CVT until its convergence limit is reached. For the generate CVT variant used, stick to Lloyd's method for logical, robust convergence, or pick a faster, superlinear method, like L-BFGS, for a great speed-up at the cost of risk of artefacts.

This strategy provides good results for boundary loops of equal shape in parallel planes, but unfortunately not in the more general case. In the search for a smooth high quality mesh, the generate CVT operator tends to eventually split up the input into two separate parts, unless the input's boundaries corresponded to a minimal surface. Its volume weight parameter can be used to combat this by preserving some volume, at least providing proper, connected results, but these are often too spherical, and hence unlike the input mesh, to be acceptable.

6.1 Future Work

Consequently, work remains to enable centroidal Voronoi tessellations to fulfil their promising role as a great new smoothing operator. First, an input mesh's optimal volume weight should be found automatically, so that one can converge to a smooth whole without human interference. But even then the operator's results will not be good enough to claim its place: they will in most cases not resemble the input enough as to appear to be a smoothed version of it.

So most importantly, seeing as the best result is usually reached before convergence has been attained, as demonstrated by the last experiment from the previous section (Figure 23), a better convergence threshold needs be invented. The smoothing operators presented are together certainly able to produce the required result of a smooth, low-cost mesh, which still resembles the initial one, but in most cases simply not up to the operators' own convergence limits. The problem is thus to find a new, external measure, which can be evaluated to detect when to stop; one that can be stated mathematically, and yet represents the intuitive notion of smoothness in the way a human observer would judge it. A solution might, for example, involve mean curvature calculations.

Thirdly, some modifications can be made to my implementations of the presented operators, to make them faster and more accurate. Generate centroidal Voronoi tessellation can in theory be sped up using a multigrid approach, with the amount of grid layers tuned to the size of the input mesh. The corresponding code could also be integrated better by removing the need for intermediate file exportation and importation, further speeding up the operator. And the quality of Laplacian smoothing can be increased by using cotangents instead of just average positions, so that geometry is actually taken into account. But ultimately, these small improvements are of minor importance for now.

On a last note, future research remains for using generate CVT in combination with the other three smoothing operators on closed meshes, in order to find the optimal strategy for those as well. Although I focussed on open meshes as these seemed more interesting to me, similar conclusions can be deduced for closed ones, as long as one takes into account the constant volume preservation necessary to refrain from converging to a single point, and to keep resembling the input after each operation. For this too, however, one needs a new, external convergence threshold. Possibly, this threshold can be one and the same.

References

[1] Hao Pan, Yi-King Choi, Yang Liu, Wenchao Hu, Qiang Du, Konrad Polthier, Caiming Zhang, and Wenping Wang. "Robust Modeling of Constant Mean Curvature Surfaces". In: *ACM Trans. Graph.* 31.4 (July 2012), 85:1–85:11. ISSN: 0730-0301. DOI: 10.1145/2185520.2185581. URL: http://doi.acm.org/10.1145/2185520.2185581.

[2] S. Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (Mar. 1982), pp. 129–137. ISSN: 0018-9448. DOI: 10.1109/TIT.1982.1056489.

[3] Qiang Du, Max Gunzburger, and Lili Ju. "Advances in Studies and Applications of Centroidal Voronoi Tessellations". In: *Numerical Mathematics: Theory, Methods & Applications* 3.2 (May 2010), pp. 119–142.

[4] Max Gunzburger Qiang Du Vance Faber. "Centroidal Voronoi Tessellations: Applications and Algorithms". In: *SIAM Review* 41.4 (1999), pp. 637–676. ISSN: 00361445. URL: http://www.jstor.org/stable/2653198.

[5] S. Fortune. "A Sweepline Algorithm for Voronoi Diagrams". In: *Proceedings of the Second Annual Symposium on Computational Geometry*. SCG '86. Yorktown Heights, New York, USA: ACM, 1986, pp. 313–322. ISBN: 0-89791-194-6. DOI: 10.1145/10515.10549. URL: http://doi.acm.org/10.1145/10515.10549.

[6] Leonidas Guibas and Jorge Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams". In: *ACM Trans. Graph.* 4.2 (Apr. 1985), pp. 74–123. ISSN: 0730-0301. DOI: 10.1145/282918.282923. URL: http://doi.acm.org/10.1145/282918.282923.

[7] M. I. Shamos and D. Hoey. "Closest-point problems". In: *Foundations of Computer Science, 1975., 16th Annual Symposium on*. Oct. 1975, pp. 151–162. DOI: 10.1109/SFCS.1975.8.

[8] P. Cignoni, C. Montani, and R. Scopigno. "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed". In: *Computer-Aided Design* 30.5 (1998), pp. 333–341. ISSN: 0010-4485. DOI: 10.1016/S0010-4485(97)00082-1. URL: http://www.sciencedirect.com/science/article/pii/S0010448597000821.

[9] H. Ledoux. "Computing the 3D Voronoi Diagram Robustly: An Easy Explanation". In: *Voronoi Diagrams in Science and Engineering, 2007. ISVD '07. 4th International Symposium on*. July 2007, pp. 117–129. DOI: 10.1109/ISVD.2007.10.

[10] A. Bowyer. "Computing Dirichlet tessellations". In: *The Computer Journal* 24.2 (1981), pp. 162–166. DOI: 10.1093/comjnl/24.2.162. eprint: http://comjnl.oxfordjournals.org/content/24/2/162.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/24/2/162.abstract.

[11] D. F. Watson. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". In: *The Computer Journal* 24.2 (1981), pp. 167–172. DOI: 10.1093/comjnl/24.2.167. eprint: http://comjnl.oxfordjournals.org/content/24/2/167.full.pdf+html. URL: http://comjnl.oxfordjournals.org/content/24/2/167.abstract.

[12] Pierre Alliez, Éric Colin De Verdière, Olivier Devillers, and Martin Isenburg. "Centroidal Voronoi diagrams for isotropic surface remeshing". In: *Graphical Models* 67.3 (2005), pp. 204–231. DOI: 10.1016/j.gmod.2004.06.007. URL: https://hal.inria.fr/hal-00787166.

[13] Dong-Ming Yan, Wenping Wang, Bruno Lévy, and Yang Liu. "Advances in Geometric Modeling and Processing: 6th International Conference, GMP 2010, Castro Urdiales, Spain, June 16-18, 2010. Proceedings". In: ed. by Bernard Mourrain, Scott Schaefer, and Guoliang Xu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Efficient Computation of 3D Clipped Voronoi Diagram, pp. 269–282. ISBN: 978-3-642-13411-1. DOI: 10.1007/978-3-642-13411-1_18. URL: http://dx.doi.org/10.1007/978-3-642-13411-1_18.

[14] D. M. Yan, W. Wang, B. Lévy, and Y. Liu. "Efficient computation of clipped Voronoi diagram for mesh generation". In: *Computer-Aided Design* 45.4 (2013), pp. 843–852. DOI: http://dx.doi.org/10.1016/j.cad.2011.09.004. URL: http://www.sciencedirect.com/science/article/pii/S0010448511002351.

[15] A. Klein, A. Certain, A. Derose, T. Duchamp, and W. Stuetzle. *Vertex-Based Delaunay Triangulation Of Meshes Of Arbitrary Topological Type.* Tech. rep. 1997.

[16] Dong-Ming Yan, Bruno Lévy, Yang Liu, Feng Sun, and Wenping Wang. "Isotropic Remeshing with Fast and Exact Computation of Restricted Voronoi Diagram". In: *Proceedings of the Symposium on Geometry Processing*. Berlin, Germany: Eurographics Association, 2009, pp. 1445–1454. URL: http://dl.acm.org/citation.cfm?id=1735603.1735629.

[17] Xiaoning Wang, Xiang Ying, Yong-Jin Liu, Shi-Qing Xin, Wenping Wang, Xianfeng Gu, Wolfgang Mueller-Wittig, and Ying He. "Intrinsic computation of centroidal Voronoi tessellation (CVT) on meshes". In: *Computer-Aided Design* 58 (2015). Solid and Physical Modeling 2014, pp. 51–61. ISSN: 0010-4485. DOI: http://dx.doi.org/10.1016/j.cad.2014.08.023. URL: http://www.sciencedirect.com/science/article/pii/S0010448514001924.

[18] Shi-Qing Xin and Guo-Jin Wang. "Improving Chen and Han's Algorithm on the Discrete Geodesic Problem". In: *ACM Trans. Graph.* 28.4 (Sept. 2009), 104:1–104:8. ISSN: 0730-0301. DOI: 10.1145/1559755.1559761. URL: http://doi.acm.org/10.1145/1559755.1559761.

[19] Y. J. Liu, Z. Chen, and K. Tang. "Construction of Iso-Contours, Bisectors, and Voronoi Diagrams on Triangulated Surfaces". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.8 (Aug. 2011), pp. 1502–1517. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2010.221.

[20] Qiang Du and Maria Emelianenko. "Acceleration schemes for computing centroidal Voronoi tessellations". In: *Numerical Linear Algebra with Applications* 13.2-3 (2006), pp. 173–192. ISSN: 1099-1506. DOI: 10.1002/nla.476. URL: http://dx.doi.org/10.1002/nla.476.

[21] James C. Hateley, Huayi Wei, and Long Chen. "Fast Methods for Computing Centroidal Voronoi Tessellations". In: *Journal of Scientific Computing* 63.1 (2015), pp. 185–212. ISSN: 1573-7691. DOI: 10.1007/s10915-014-9894-1. URL: http://dx.doi.org/10.1007/s10915-014-9894-1.

[22] Lin Lu, Bruno Lévy, and Wenping Wang. "Centroidal Voronoi Tessellation of Line Segments and Graphs". In: *Computer Graphics Forum* 31.2 (2012), pp. 775–784. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2012.03058.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2012.03058.x.

[23] Yang Liu, Wenping Wang, Bruno Lévy, Feng Sun, Dong-Ming Yan, Lin Lu, and Chenglei Yang. "On Centroidal Voronoi Tessellation - Energy Smoothness and Fast Computation". In: *ACM Trans. Graph.* 28.4 (Sept. 2009), 101:1–101:17. ISSN: 0730-0301. DOI: 10.1145/1559755.1559758. URL: http://doi.acm.org/10.1145/1559755.1559758.

[24] Dong C. Liu and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization". In: *Mathematical Programming* 45.1 (1989), pp. 503–528. ISSN: 1436-4646. DOI: 10.1007/BF01589116. URL: http://dx.doi.org/10.1007/BF01589116.