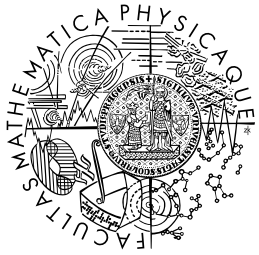



FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

**MASTER THESIS**

Bc. Michal Ivičič

# Feature preserving triangle mesh fairing

Department of Software and Computer Science Education

Supervisor of the master thesis: Ing. Vojtěch Bubník

Study programme: Computer Science

Study branch: Computer Graphics and Game Development

Prague 2023

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....  .....

Author's signature

I would like to thank my supervisor Ing. Vojtěch Bubník for his feedback and for allowing me to work on this for me interesting task. Next I would like to make a general thank to all university professors who put the recordings of their lectures on the internet for people to see. Finally I thank my friends and family for their support.

Title: Feature preserving triangle mesh fairing

Author: Bc. Michal Ivičič

Department: Department of Software and Computer Science Education

Supervisor: Ing. Vojtěch Bubník, Department of Software and Computer Science Education

Abstract: In this thesis, we address the problem of smooth interpolatory subdivision of triangle meshes. This problem is more challenging than general smooth subdivision as the original vertices of the mesh cannot be moved to produce smoother surfaces. There are also not that many existing works that cover this problem. As a solution, we use the minimization of a global fairness energy to compute the positions of newly added vertices of the mesh. This approach results in smoother surfaces than the widely used butterfly subdivision scheme, but at the cost of increased computational complexity and potential volume loss in the resulting mesh. We analyze the sources of volume loss and implement a solution to prevent it. Our algorithm also preserves sharp edges and points on the mesh surface. While our solution performs well for basic objects, it may produce artifacts for more complex ones. As a future work, we suggest ways to improve our algorithm to address the artifact occurrences.

Keywords: discrete differential geometry, mesh processing, subdivision

# Contents

# Introduction

## Background and motivation

Three-dimensional (3D) models are widely used in industries such as animation, game development, architecture, and product design. These models are usually created using 3D modeling software by artists or designers and are used for purposes such as visualization, 3D printing, or scientific simulations. Some models can also be obtained through real-world scanning. This thesis solves a specific problem that involves the processing of 3D models.

In this section, we provide a high-level overview of the problem of this thesis along with some background information. More detailed definitions of some of the mentioned terms follow in later chapters.

### Subdivision surfaces

In general, the task of this thesis is to construct an algorithm that takes as input a 3D model represented by a *triangle mesh*. The algorithm then produces another mesh consisting of more triangles that should resemble a more detailed version of the input. This technique is known as *subdivision surfaces* and typically requires two steps. First, the edges and faces of the original mesh are divided into smaller parts (e.g., each edge is split into two and each triangle into four smaller ones). This also involves the addition of new vertices. Second, the vertices of the new mesh are moved to make the new surface smoother. The smoothing of a vertex is commonly performed by taking a weighted average of neighboring vertices' positions. If a method moves only the inserted vertices and keeps the positions of the original ones fixed, it is called *interpolatory*. Interpolatory methods have an additional constraint and are generally more difficult to optimize for good results.

A common workflow exists in 3D modeling, where an artist first edits a coarser mesh, where larger changes in the model's shape are easier to make. The changes to the coarser mesh typically involve moving only one or a few vertices or adding new faces next to existing ones. The model can then be subdivided into a finer mesh, where the artist can work locally on higher-resolution details. A mesh can be subdivided this way multiple times during the modeling process.

Another application is to only model or acquire the coarse mesh and let an algorithm generate a finer mesh without any user input along the way. This might be useful for fast prototyping or in 3D printing to increase the smoothness of a model. During the process, a large number of new vertices are added to the model at once, so there is a higher emphasis on the quality of the subdivision algorithm. Subdivision schemes that only average neighboring vertices' positions may not suffice for this purpose. There is also a need to somehow preserve *sharp features* of the original mesh—sharp edges and vertices (e.g., sides of a cube or a vertex on top of a cone).

## How to evaluate a subdivision

One way to evaluate the quality of a subdivided triangle mesh is to use the notion of *smoothness* or *fairness*—the definitions of these terms often vary across applications. One notion of smoothness might be a certain degree of *geometric continuity* greater than zero (e.g., *tangent continuity* denoted as $G^1$). Triangle meshes are piece-wise linear surfaces and are therefore never smooth in the geometric sense. In some subdivision approaches, new vertices are constructed to be on a parametrically defined surface or curve, which in fact is smooth in the sense of geometric continuity (e.g., polynomial patches or curves). The difference[1] between the new mesh and the parametric object could, in a limit, approach zero as we increase the number of new vertices in the subdivision. Then we might say a part of a triangle mesh is also smooth as it *approximates* a smooth parametric object.

The notion of fairness may, in some cases, mean only looking *aesthetically pleasing* or with no *unnatural features* (without any formal definition). This is a valid criterion, as subdivided meshes might, in some cases, end up directly in rendered images, videos, or other content. In this thesis, we extensively employ this informal approach for evaluation, as we aim to avoid the generation of noticeable visual artifacts by a corresponding algorithm.

The book *Polygon Mesh Processing* [Botsch et al., 2010] gives the following description of fairness: "... in general fair surfaces should follow the *principle of simplest shape*: the surface should be free of any unnecessary details or oscillation." One way to achieve fairness in practice is by minimizing a defined energy function dependent on vertex positions across the mesh. This may involve minimizing the total surface area or discretized surface curvature along with constraints for the positions of certain vertices. The methods that optimize the energy functions operate on the whole mesh and are therefore more computationally expensive. On the other hand, no unnecessary oscillation is produced, as might be the case for a simple average of neighboring vertices' positions.

Another approach to evaluating a subdivision (common in machine learning) is to have a dataset of pairs of meshes—a coarse and a fine version of the same object. Different subdivision algorithms could be compared by looking at the differences between the original fine mesh and a mesh subdivided from the coarse mesh using a particular mesh distance metric. This approach is used by Liu et al. [2020] for evaluation and training of their *neural subdivision*. Public datasets[2] are available to be used for this evaluation method; if the coarser meshes are not already a part of the dataset, they might be generated by a suitable mesh simplification algorithm, such as *QSLIM* [Garland and Heckbert, 1997]. It is important to note that if some features, such as high-frequency details, are lost during the creation of coarser meshes, there is no way for the energy minimizing or vertex averaging algorithms to reconstruct them back. This can only be achieved by prediction algorithms over training datasets or by rule-based manual addition of high-frequency details based on coarse meshes' features.

[1] *Hausdorff distance* may be used for computing the difference between two objects [Bartoň et al., 2010].

[2] For example the *ABC* dataset of Computer-aided design (CAD) models [Koch et al., 2019].

## Additional background

A two-dimensional (2D) analogy of triangle mesh subdivision might be image super-resolution, which attempts to construct a higher-resolution image from a lower-resolution one. The state of the art solutions for this task are now using *deep neural networks* [Wang et al., 2020].

We can also look again at the process of creating a finer mesh from a coarser one without any user input along the way. An analogous process in 2D is using a coarse image drawing as an input to a diffusion model such as Stable Diffusion [Rombach et al., 2022] and getting as an output a more detailed image resembling the input. Apart from the optional coarse image input, a more important type of input to a diffusion model is a text prompt that may describe a desired object in the resulting picture or an artistic style.

Attempts to generate 3D models from text prompts have also been made, with one of the most successful ones being Magic3D [Lin et al., 2023]. It generates the final 3D model in a coarse to fine manner, where a model is initially represented as a neural network encoding [Müller et al., 2022] and in the finer stages as a tetrahedral mesh. The model is iteratively being improved, where in each iteration it is rendered from a certain angle to a 2D texture, that is given as an input to a 2D diffusion model along with a text prompt. The diffusion model generates normal, color, and other types of textures that are used to update the 3D model.

This thesis is about interpolatory subdivision, which is covered slightly less in terms of produced algorithms than the general case. Interpolatory subdivision may be beneficial, for example, in 3D printing, where it is in some cases important to keep the outlines of models fixed (e.g., in mechanical parts that later need to fit together). While not using machine learning or neural networks, the proposed algorithm intends to combine an energy-minimizing global fairing approach with the retention of sharp features of the input mesh.

The requirements we set for the algorithm are the following:

1. The algorithm should perform interpolatory subdivision on triangle meshes with the amount of inserted new vertices somehow controlled by input parameters. The input mesh is assumed to be a manifold (see section 1.2.1) and the output mesh should be a manifold as well.

2. The algorithm should be able to preserve the sharp features of the input mesh.

3. The algorithm should, to some degree, satisfy either fairness or resemblance to an original finer mesh.

4. There should be a reasonable tradeoff between the speed and quality of the output mesh.

While we present our algorithm as a whole, there are parts of it with various degrees of significance or replaceability. We can categorize them as follows (although the algorithm features are more on a spectrum between the categories than strictly adhering to one category):

**Strictly set features** The parts with clearly defined, non-ambiguous ways to implement them.

**General problems** This is, for example, sharp edge detection. A problem for which many different approaches exist. The more optimal approaches are often more complicated to implement. Because of the independent nature of the general problem, our algorithm can be easily improved by swapping a usually simpler solution to the problem proposed by us for a more robust solution.

**Approximative methods for specific problems** These are our solutions to smaller problems that arise during our proposed algorithm. For example, the computation of a vector that is most close to being perpendicular to a set of other vectors (see section 3.4). What we aim for is a rough approximation that is efficient to compute. Our solutions to these kinds of problems are usually simple, non-optimal, and sufficient to work in our case. The solutions should not be taken as something essential to our algorithm and could be exchanged for different approximation methods solving the same task.

The overall structure of the thesis is as follows:

1. **Geometric background** We start with a brief survey of parts of continuous differential geometry, with an emphasis on surface curvature. Then we move to the discrete world, where we show how analogous concepts to the continuous ones can be defined for triangle meshes.

2. **Related works** In this section, we provide a brief overview of various approaches to interpolatory and non-interpolatory subdivision and smoothing. This includes traditional subdivision schemes that use neighborhood averaging and a fairing method known as curvature flow. We also explore the use of neural networks to address the problem. Additionally, we discuss polynomial interpolation, which could potentially serve as a subdivision scheme. Some of the covered polynomial interpolation schemes preserve sharp features.

3. **Our approach** In chapter 3 we present our algorithm for interpolatory subdivision. We start with an overview of an existing energy minimization fairing approach. We show that energy minimization is undesirably shrinking sections of the mesh that resemble parts of a cylinder and propose a countermeasure to the problem. In addition, we describe how to perform feature detection on the coarse mesh, which results in adjustments to the energy minimization. That involves, among other things, the polynomial and linear interpolation of curves and surfaces near sharp edges and vertices.

4. **Results and evaluation** In this chapter, we measure the speed of our algorithm and compare its smoothing capabilities to butterfly subdivision. We also show the results of our method for different non-smooth shapes and cover problematic cases that occur for certain types of models.

5. **Future work** We discuss possible ways in which our algorithm might be improved—we address the problematic cases of our approach from the previous chapter and propose ways to avoid them.

# 1. Geometric background

## 1.1 Differential geometry

In this section, we present a brief survey of parts of continuous differential geometry, focusing on surface curvature. The main purpose of this section is to define concepts that will be referenced in later parts of the text. For proofs, intuitions behind the definitions, and a more detailed treatment of the subject, we encourage the reader to seek other sources, such as Kühnel [2015]. Main references for this section are lecture notes of an MIT course on *shape analysis* from Solomon [2021] and a book by Botsch et al. [2010].

### 1.1.1 Submanifolds

In our first definition, we introduce the notion of a *submanifold* of $\mathbb{R}^n$. This will be the basic object that allows us to measure curvature, distances, and other properties upon it.

**Definition 1** (Submanifold of $\mathbb{R}^n$)**.** *A set $\mathcal{M} \subseteq \mathbb{R}^n$ is an m-dimensional submanifold of $\mathbb{R}^n$ if for each $\boldsymbol{p} \in \mathcal{M}$ there exist open sets $\mathcal{U} \subseteq \mathbb{R}^m$, $\mathcal{V} \subseteq \mathbb{R}^n$ and a function $g\colon \mathcal{U} \to \mathcal{M} \cap \mathcal{V}$ such that $\boldsymbol{p} \in \mathcal{V}$ and $g$ is an infinitely differentiable ($C^\infty$) bijection with a Jacobian of rank $m$ and a continuous inverse function.*

The function $g$ is generally referred to as a *map* and $\mathcal{U}$ as a *parameter space*. In some cases, the infinite differentiability of $g$ is exchanged for less or more strict differentiability constraints.

One-dimensional submanifolds of $\mathbb{R}^2$ or $\mathbb{R}^3$ are *curves*, whereas two-dimensional submanifolds of $\mathbb{R}^3$ are *surfaces*. A common way to define these objects is by *parametric representation*. In the following equation, we present as an example a parametric representation of a sphere with radius $r$ and input parameters $\theta \in [0, 2\pi)$ and $\phi \in [0, \pi]$:

$$f(\theta, \phi) = \begin{pmatrix} r \sin\phi \cos\theta \\ r \sin\phi \sin\theta \\ r \cos\phi \end{pmatrix}.$$

The function $f$ from the equation above can be used as the map $g$ in Definition 1, except for the boundary points for $\theta$ and $\phi$, where $f$ does not have a continuous inverse function. In these cases, we would need to use different local parametrization for $g$.

### 1.1.2 Curves

In this subsection about curves, we first show how to make a curve *parametrized by arc length*. This ensures that the derivative of the curve function is always unit length, which allows us to define *tangents* and *normals* for curves in $\mathbb{R}^3$.

For a given point $t_0$, the length of a curve parametrized by $\gamma$ between the points $\gamma(t_0)$ and $\gamma(t)$ is given by the integral

$$S(t) := \int_{t_0}^{t} \|\gamma'(s)\|_2 \, ds.$$

The notation $\|\cdot\|_2$ represents the Euclidean norm of a vector. As the function $S$ is *strictly increasing*, it is possible to define $\tilde{\gamma}(t) := \gamma(S^{-1}(t))$ as the composition of the inverse of $S$ and $\gamma$. This operation is known as *reparametrization by arc length* and can be done for every curve. It can be shown that for all points $s$: $\|\tilde{\gamma}'(t)\|_2 = 1$.

Suppose $\gamma\colon (a, b) \to \mathbb{R}^3$ is a curve parametrized by arc length and for all $s \in (a, b)$: $\gamma''(s) \neq 0$. We define the unit tangent of $\gamma$ as $\boldsymbol{T}(s) := \gamma'(s)$. The unit normal of $\gamma$ is defined as $\boldsymbol{N}(s) := \gamma''(s)/\|\gamma''(s)\|_2$. Together with *curvature* $\kappa := \|\gamma''(s)\|_2$, this gives us the formula:

$$\boldsymbol{T}'(s) := \kappa(s)\boldsymbol{N}(s).$$

The curvature expresses a notion of how the tangent of the curve changes with respect to the arc length. If we take, for example, a circle in 3D with a radius of $r$, its curvature is everywhere $1/r$. As the radius of the circle increases, the circle's curve starts to locally resemble a line, and the curvature approaches zero.

In addition to curvature, there is also *torsion* which together with curvature defined for all points is enough to determine the shape of the whole curve up to rigid motion (translation, rotation, and reflection).

### 1.1.3 Surfaces

In this subsection, we define the basic properties of surfaces, including *first and second fundamental forms*, *Gaussian and mean curvature* and *geometric continuity.*

#### Tangent and normal spaces

We will refer to the next definition also as a *tangent plane.*

**Definition 2** (Tangent space of a surface)**.** *The tangent space $T_{\boldsymbol{p}}\mathcal{S}$ of a surface $\mathcal{S}$ at a point $\boldsymbol{p}$, where $g(u, v)\colon \mathbb{R}^2 \to \mathbb{R}^3$ is the map $g$ form Definition 1, and $g(u_0, v_0) = \boldsymbol{p}$, is given by*

$$T_{\boldsymbol{p}}\mathcal{S} := \operatorname{span}\left\{\frac{\partial g}{\partial u}(u_0, v_0), \frac{\partial g}{\partial v}(u_0, v_0)\right\}. \tag{1.1}$$

The *span* operator represents the linear span of vectors from the corresponding set. The linear independence of the two partial derivatives of $g$ is attributed to the rank of the Jacobian of $g$, which has a value of 2. Equation 1.1 can be simplified to

$$T_{\boldsymbol{p}}\mathcal{S} := \operatorname{span}\{\boldsymbol{p}_u, \boldsymbol{p}_v\}.$$

We will use this subscript notation throughout the rest of the text; it will always mean the partial derivatives of function $g$ taken from Definition 1 at point $\boldsymbol{p}$. Similarly, we denote $\boldsymbol{p}_{uu}$, $\boldsymbol{p}_{vv}$ and $\boldsymbol{p}_{vu}$ as the corresponding second partial derivatives of $g$ at $\boldsymbol{p}$.

Finally, we denote $\boldsymbol{p}_{\boldsymbol{w}}$, where $\boldsymbol{w} = (w_u, w_v)^T$, the *directional derivative* of $g$ at $\boldsymbol{p}$ in the direction $\boldsymbol{w}$, which is given by

$$\boldsymbol{p}_{\boldsymbol{w}} = \frac{\partial g}{\partial u}(u_0, v_0)w_u + \frac{\partial g}{\partial v}(u_0, v_0)w_v.$$

The directional derivative can give us a sense of how the surface changes in the direction $\boldsymbol{w}$ by returning values in the tangent plane $T_{\boldsymbol{p}}\mathcal{S}$.

**Definition 3** (Normal space of a surface)**.** *The normal space $N_{\boldsymbol{p}}\mathcal{S}$ of a surface $\mathcal{S}$ at point $\boldsymbol{p}$ is defined as*

$$N_{\boldsymbol{p}}\mathcal{S} := \operatorname{span}\{\boldsymbol{p}_u \times \boldsymbol{p}_v\}.$$

The symbol $\times$ represents the operator for the cross product of two vectors. If we want to select a normal vector $\boldsymbol{n}_p \in N_{\boldsymbol{p}}\mathcal{S}$ of a tangent plane $T_{\boldsymbol{p}}\mathcal{S}$, we have two options—the normalized versions of $\boldsymbol{p}_u \times \boldsymbol{p}_v$ and $-\boldsymbol{p}_u \times \boldsymbol{p}_v$. We might try to construct a continuous function $f$ that maps each point on the surface to its normal. This is only possible for the so called *orientable*[1] surfaces. From now on, we assume that all surfaces mentioned in this text are orientable, and therefore the normals $\boldsymbol{n}_p$, defined by $f$, are available for all points $\boldsymbol{p}$ on the surface.

### First and second fundamental form

**Definition 4** (First fundamental form)**.** *The first fundamental form $\mathbf{I}_{\boldsymbol{p}}$ of a surface $\mathcal{S}$ at point $\boldsymbol{p}$ is given by*

$$\mathbf{I}_{\boldsymbol{p}} := \begin{pmatrix} \boldsymbol{p}_u^T\boldsymbol{p}_u & \boldsymbol{p}_u^T\boldsymbol{p}_v \\ \boldsymbol{p}_v^T\boldsymbol{p}_u & \boldsymbol{p}_v^T\boldsymbol{p}_v \end{pmatrix}.$$

The form can be viewed as an operator

$$\mathbf{I}_{\boldsymbol{p}}(\boldsymbol{w}_1, \boldsymbol{w}_2) := \boldsymbol{w}_1^T\mathbf{I}_{\boldsymbol{p}}\boldsymbol{w}_2$$

defined for each surface point. It can be shown that the operator returns a dot product between the directional derivatives of $\boldsymbol{p}$ in directions $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$. From the dot product defined as such, it is possible to measure the size of vectors in the tangent space and the angles between them. More advanced uses of the first fundamental form involve the computation of the length of a parametrized surface curve or the area of a parametrized surface patch.

**Definition 5** (Second fundamental form)**.** *The second fundamental form $\mathbf{II}_{\boldsymbol{p}}$ of a surface $\mathcal{S}$ at point $\boldsymbol{p}$ is given by*

$$\mathbf{II}_{\boldsymbol{p}} := \begin{pmatrix} \boldsymbol{p}_{uu}^T\boldsymbol{n}_{\boldsymbol{p}} & \boldsymbol{p}_{uv}^T\boldsymbol{n}_{\boldsymbol{p}} \\ \boldsymbol{p}_{uv}^T\boldsymbol{n}_{\boldsymbol{p}} & \boldsymbol{p}_{vv}^T\boldsymbol{n}_{\boldsymbol{p}} \end{pmatrix}.$$

[1] An example of a surface that is not orientable is the Klein bottle [Polthier, 2003].

The second fundamental can be seen as an operator

$$\mathbf{II}_{\boldsymbol{p}}(\boldsymbol{w}_1, \boldsymbol{w}_1) := \boldsymbol{w}_1^T \mathbf{II}_{\boldsymbol{p}} \boldsymbol{w}_2.$$

An important application of the second fundamental form is to measure the curvature of a curve along the surface $\mathcal{S}$. A surface curve might be locally defined as a 2D curve in parameter space that is mapped to the surface space using function $g$ from definition 1. The curvature of a surface curve at point $\boldsymbol{p}$ depends only on the tangent $\boldsymbol{t}$ of the corresponding 2D curve in parameter space at a point corresponding to $\boldsymbol{p}$. For all surface curves $\gamma_{\boldsymbol{t}}$ with the parameter space tangent $\boldsymbol{t}$ at $\boldsymbol{p}$, it holds that

$$\mathbf{II}_{\boldsymbol{p}}(\boldsymbol{t}, \boldsymbol{t}) = \kappa(s)\boldsymbol{N}(s),$$

where $\kappa(s)$ is the curvature of $\gamma_{\boldsymbol{t}}$ at $\boldsymbol{p}$, and $\boldsymbol{N}(s)$ is the normal vector of $\gamma_{\boldsymbol{t}}$ at $\boldsymbol{p}$.

### Mean and Gaussian curvature

As $\mathbf{II}_{\boldsymbol{p}}(\boldsymbol{w}_1, \boldsymbol{w}_1)$ is a symmetric matrix, it has real eigenvalues, which we denote $\kappa_{\min}$ and $\kappa_{\max}$, where $\kappa_{\min} \leq \kappa_{\max}$. We refer to them as the *principal curvatures* of the surface $\mathcal{S}$ at point $\boldsymbol{p}$. They are, respectively, the minimum and maximum values of the curvature of a curve along a surface $\mathcal{S}$ at point $\boldsymbol{p}$. The eigenvectors (tangent vectors in the parameter space) corresponding to $\kappa_{\min}$ and $\kappa_{\max}$ are orthogonal to one another.

**Definition 6** (Mean and Gaussian curvature)**.** *The mean curvature $H$ and Gaussian curvature $K$ of a surface $\mathcal{S}$ at point $\boldsymbol{p}$ are defined as*

$$H := \frac{1}{2}\left(\kappa_{\min} + \kappa_{\max}\right),$$
$$K := \kappa_{\min}\kappa_{\max}.$$

A point $\boldsymbol{p}$ with Gaussian curvature $K$ can be categorized into three cases (as can be seen in figure 1.1.3):

(a) If $K > 0$, the surfaces locally resemble valleys or the tops of hills; we call these points *elliptic*.

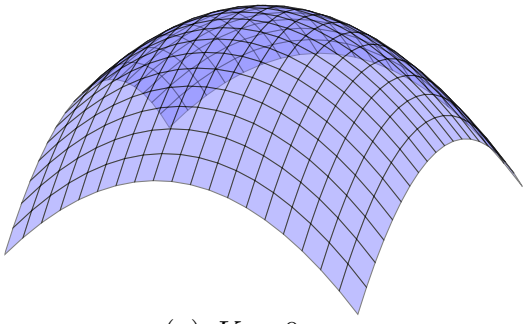
(b) If $K = 0$, the surface is locally flat, like a plane, or bent in only one direction. These are *parabolic* points. Apart from the two other cases, the surface is locally not being stretched.

(c) If $K < 0$, the surface is locally saddle-shaped, we call these points *hyperbolic*.

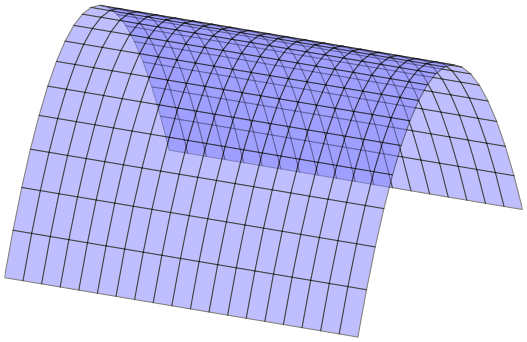
### Geometric continuity

We have now introduced enough concepts to define the notions of *geometric* continuity.

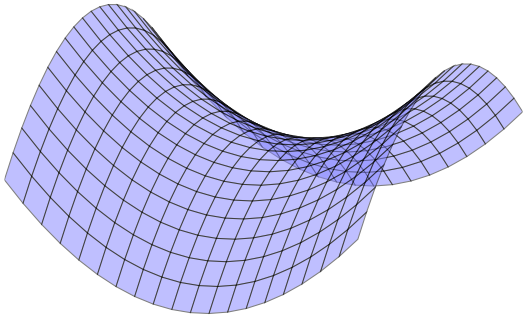
**Definition 7** (Geometric continuity)**.** *The $G^0$, $G^1$ and $G^2$ geometric continuity of surfaces are defined as follows:*



(a) $K > 0$



(b) $K = 0$



(c) $K < 0$

Figure 1.1: Examples of surfaces with different Gaussian curvature.

- *A surface $\mathcal{S}$ is $G^0$-continuous if it is a submanifold of $\mathbb{R}^n$ with a continuous function $g$ from Definition 1.*
- *A surface $\mathcal{S}$ is $G^1$-continuous if it has a continuous tangent plane $\mathcal{T}_{\boldsymbol{p}}$ at every point $\boldsymbol{p}$.*
- *A surface $\mathcal{S}$ is $G^2$-continuous if it has a continuous second fundamental form $\mathbf{II}_{\boldsymbol{p}}$ at every point $\boldsymbol{p}$.*

## 1.2 Discrete geometry

In this section, we cover how to apply the concepts of curvature and other properties of surfaces to triangle meshes. Main references for this section are Meyer et al. [2003], Solomon [2021] and Botsch et al. [2010].

### 1.2.1 Triangle meshes

#### General definitions

The following definitions will build into an idea of a *simplical complex*. Simplical complex is a general term that encompasses triangle meshes and, among others, also *graphs* (meaning a collection of vertices and edges) and meshes in higher dimensions (e.g., *tetrahedral meshes*).

**Definition 8** (Simplex)**.** *A $k$-simplex is a set $S \subseteq \mathbb{R}^n$ that for some affinely independent set of points $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k \in \mathbb{R}^n$ satisfies*

$$S = \left\{ \boldsymbol{r} \in \mathbb{R}^n \;\middle|\; \boldsymbol{r} = \sum_{i=0}^{k} \alpha_i \boldsymbol{p}_i \text{ where } \sum_{i=0}^{k} \alpha_i = 1 \text{ and } \forall i : \alpha_i \geq 0 \right\}.$$

We denote $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k$ as vertices of $S$. Simplices of dimensions 0 to 3 are points, lines (edges), triangles, and tetrahedra, respectively.

**Definition 9** (Faces of a simplex)**.** *The faces of a simplex given by vertices $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k \in \mathbb{R}^n$ are all of the $l$-simplices ($l \leq k$) defined by a subset of the vertices $\boldsymbol{p}_0, \ldots, \boldsymbol{p}_k$, including the empty set.*

For example, the faces of a triangle are the three points and lines that define the triangle, with the addition of the empty set and the triangle itself.

**Definition 10** (Simplical complex)**.** *A simplical complex $\mathcal{K}$ is a set of simplices such that for all $S \in \mathcal{K}$, the faces of $S$ are also in $\mathcal{K}$, and for any two simplices $S, S' \in \mathcal{K}$, the intersection $S \cap S'$ is a face of both $S$ and $S'$.*

We consider simplices $S$ and $S'$ $k$-adjacent if $S \cap S'$ is a simplex of dimension $k$. To avoid confusion in this text, if we omit the $k$ and say the simplices are only adjacent, we mean $(l-1)$-adjacency if both simplices have dimension $l$, and in the case of different dimensions, we mean $l$-adjacency where $l$ is the dimension of the simplex with the lower dimension. From this, it follows that two triangles are adjacent if they share an ege, and a triangle is adjacent to a vertex if it contains the vertex.

It is possible to define the concept of discrete manifolds for all simplical complexes. We will however restrict ourselves only to *triangle meshes*, which for us are simplical 2-complexes (the maximum dimension of a simplex is 2) with points in $\mathbb{R}^3$. In order to be a discrete manifold, a 2-complex has to satisfy the following properties:

- Every edge is adjacent to exactly two triangles. For a triangle mesh with a boundary, it is sufficient to be adjacent to one triangle.
- For all vertices $v$, the set of all triangles that have $v$ as a vertex forms a *closed fan* (or an *open fan* for a mesh with a boundary).

A *fan* is a sequence of different triangles that all share a common vertex, and every two consecutive triangles share an edge. The fan is closed if the last triangle shares an edge with the first triangle; otherwise, it is open. If we remove the condition of the common vertex from the open fan sequence, we get a *generalized triangle strip*. By imposing a condition on the generalized triangle strip that at most three triangles share a common vertex, we get a (standard) *triangle strip*. Figure 1.2.1 shows examples of the defined triangle formations.

We denote the neighborhood $\mathcal{N}(v)$ of a vertex $v$ as the set of all vertices distinct from $v$ that share a common edge with $v$. The degree of a vertex $v$ is defined as $\deg(v) = |\mathcal{N}(v)|$.

A vertex is *regular* if it has a degree of 6. A mesh with a very high amount of regular vertices (*highly regular mesh*) has in general triangles of interior angle degrees close to 60—the triangles are close to being equilateral. This is a desirable property for mesh processing. The cotangent Laplacian operator, which we introduce later, works better for a mesh with triangles close to being equilateral.

The normal of a triangle is the normalized cross product between two edges of the triangle. As in the continuous case, however, we have two choices to define it uniquely because the negative of the normal is also a valid choice. We can make the triangles of the mesh *oriented* by assigning to each of them an *orientation*—an ordered sequence of its vertices $(v_1, v_2, v_3)$. Then we can define the unique normal of a triangle as

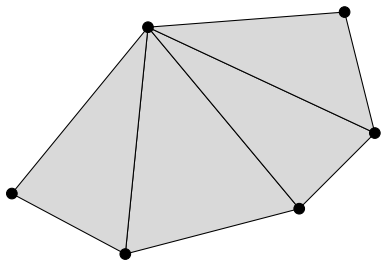
$$\boldsymbol{n} = \frac{(\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)}{\|(\boldsymbol{p}_2 - \boldsymbol{p}_1) \times (\boldsymbol{p}_3 - \boldsymbol{p}_1)\|_2},$$

where $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$ are the positions of vertices $v_1, v_2, v_3$, respectively.

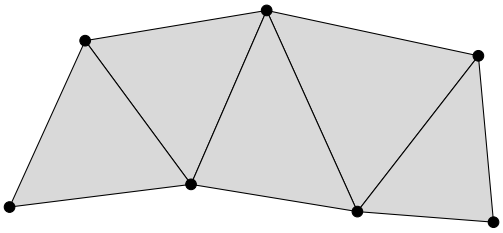
To make the orientation consistent across the whole mesh, we require that if two triangles share a common edge $\{v_1, v_2\}$, then the vertices $v_1, v_2$ appear in opposite order in the two triangles' orientations. One of the triangles then has orientation $(v_1, v_2, v_3)$, and the other $(v_2, v_1, v_4)$, up to cyclic permutations.

The computation of a normal at a vertex $v$ of a triangle mesh is generally done by averaging the normals of triangles that are in a closed fan around $v$. A weighting scheme can be employed for the averaging. The most common one is *angle weighted averaging*, where the normal of a triangle $t$ is weighted by the size of the interior angle of $t$ next to $v$.

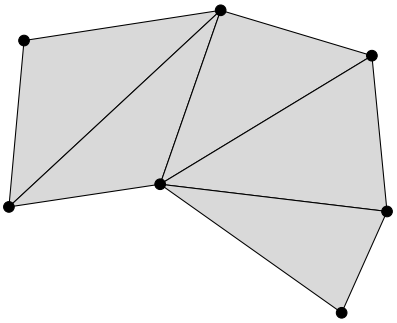
The *dihedral angle* of a triangle mesh is the angle between two adjacent triangles sharing an edge. More precisely, the dihedral angle of an edge $e$ of a triangle



(a) Triangle fan



(b) Triangle strip



(c) Generalized triangle strip

Figure 1.2: Examples of triangle formations.

mesh is the angle between the normals of the two triangles adjacent to $e$. The dihedral angles, in a way, define the curvature of a curve running along the triangle mesh.

#### Representations

A simple format to store an oriented triangle mesh is the *shared vertex representation*. It contains two lists—one for vertices and one for triangles. The vertex list contains the position of each vertex in $\mathbb{R}^3$, which is given by three floating point numbers. Each triangle is defined by an array of three indices into the vertex list, which also defines the orientation of the triangle. This representation is very compact and is used by the popular `.obj` file format. However, it does not allow for easy traversal of the mesh. We cannot, for example, find the neighboring triangles of a triangle in constant time.

A more convenient representation is the *half-edge representation*, where *half-edge* is a pair $(e, f)$ of an edge $e$ and a triangle $f$ adjacent to it. Given one half-edge, the other half-edge with $e$ called the *opposite* should be accessible in constant time, as well as the *next* and *previous* (according to triangle orientation) half-edges of the triangle $f$. A half-edge can also be viewed as pointing from one vertex to another, these two vertices being in the same order as in the orientation of $f$. The positions of the vertices of the half-edge should also be accessible in constant time. This is commonly implemented using pointers to objects. An alternative way is to use the two lists for vertices and triangles from shared vertex representation, where each triangle has three additional indices back into the triangle list that define its three neighbors. In this approach, a half-edge is a pair of an index to the triangle list and a number from 0 to 2 that defines the edge of the triangle. With the right correspondence between edge numbers and neighboring triangles' indices, it is possible to perform the previously mentioned traversal operations in constant time.

### 1.2.2 Voronoi regions

*Voronoi regions* are a partition of a given space that assigns to each point $\boldsymbol{p}_i$ from $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$ all points of the space that are closer to $\boldsymbol{p}_i$ than to any other vertex from $\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n$. In $\mathbb{R}^2$ this is known as the *Voronoi diagram*. In the case of a triangle mesh, the Voronoi region of a vertex $v$ is the set of points that are closer to $v$ than to any other vertex of the mesh (not in $\mathbb{R}^3$ but using *geodesic distance*, which is the length of the shortest path going only over the triangle mesh surface). We can compute the area of the Voronoi region of $v$ as the sum of areas of closest points to $v$ in each triangle adjacent to $v$ (with a small modification).

In a triangle, a point equidistant from all of its three vertices is the *circumcenter*, which is also the point where the three Voronoi regions corresponding to the vertices of the triangle meet. In the case of an *obtuse* triangle, however, the circumcenter is located outside of the triangle. This means that in some cases, the Voronoi region of a vertex can span beyond its adjacent triangles. Therefore, instead of computing the true Voronoi regions, we partition the obtuse triangle such that the three modified Voronoi regions meet in the middle of the longest edge of the triangle. In this partition of an obtuse triangle $T$, we set the area of the partition belonging to the vertex opposite to the longest edge to be $A(T)/2$,

and to the other two vertices we assign the area value $A(T)/4$ (the value $A(T)$ represents the area of $T$).

If a triangle $ABC$ is *acute*, the area of the Voronoi region corresponding to $A$ can be computed as $\frac{1}{8}(|AB|^2 \cot \angle C + |AC|^2 \cot \angle B)$ [Meyer et al., 2003]. In combination with the definitions of area for Voronoi regions of obtuse triangles, we can estimate the area of the Voronoi region of a vertex $v$ as the sum of the areas of the Voronoi regions of $v$ in the triangles adjacent to $v$. We denote the result as the *mixed Voronoi area* of $v$.

### 1.2.3 Discrete curvature

In the context of discrete differential geometry, the *discretizations* of continuous concepts, such as the curvature of a curve applied to a piece-wise linear discrete curve, often lead to multiple definitions. While there are many desirable properties in the continuous case that we would like to preserve, it is not always possible to retain all of them while using a given discretization scheme. For the curvature of curves, it is not possible to simultaneously preserve formulas for the curve's *winding number* and the variation of a curve's *arc length*, which both use continuous curvature (definitions for these concepts can be found in Solomon [2021]). This issue is of great importance in the field of geometric processing, where the choice of a discretization scheme can have a significant impact on the quality of the obtained results.

#### Gaussian curvature

The following definition of Gaussian curvature of a vertex $v_i$ is derived from the preservation of a theorem called *Gauss-Bonet* for triangle meshes [Solomon, 2021]:

$$K = \frac{1}{A_i}\left(2\pi - \sum_j \theta_j\right),$$

where $A_i$ is the mixed Voronoi area of $v_i$ and the sum over $j$ iterates over all triangles adjacent to $v_i$ and sums their interior angles corresponding to $v_i$. By taking the previous equation without the division by $A_i$, we obtain the *integrated Gaussian curvature* of $v_i$.

#### Laplace-Beltrami operator

In conclusion of our discussion on discrete geometry, we introduce the *cotangent Laplacian*, which provides a discretization of the *Laplace-Beltrami* operator for triangle meshes. The *Laplace-Beltrami* operator is a generalization of the *Laplace operator* to surfaces. The Laplace operator $\Delta$ for multi-variable functions is defined as the divergence of the gradient: $\Delta f = \nabla \cdot \nabla f$ (the symbol $\cdot$ represents the dot product). For a function of two variables $f(x, y)$ this can be written as

$$\Delta f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}.$$

The intuition behind the Laplace operator is that it, in a sense, captures how a point differs from an average value over its closest neighborhood. If the neighborhood is locally flat, like a plane, the Laplace operator is zero.

The Laplace-Beltrami operator $\Delta_{\mathcal{S}}$ for surfaces belongs to more advanced topics in differential geometry, and in this text we have not introduced enough theory to provide an exact definition for it—it can be found in Jost and Jost [2008]. The Laplace-Beltrami operator does not depend on surface curvature and can be computed entirely from the first fundamental form.

The Laplace operator is also featured in many applications in physics, such as the *heat equation* and the *wave equation*. These equations can also be applied to manifolds. A lot of the recent progress in the fields of discrete differential geometry and shape analysis is due to the application of the Laplacian to triangle meshes and other discrete structures. For instance, there exists the concept of *manifold harmonics*, which can be used to measure the frequencies of possible waves on manifolds and, in turn, on triangle meshes. The computed frequencies can carry a lot of information about the shape of a surface [Vallet and Lévy, 2008]. Another different application of the Laplacian is using the heat equation to compute distances on meshes in Crane et al. [2013]. In the case of this thesis, we will examine the use of the Laplace-Beltrami operator for surface smoothing.

Many different discretizations of the Laplace-Beltrami operator to triangle meshes exist. Similarly to discrete curvature, there is not a discretization scheme the would preserve all significant properties of the continuous operator [Wardetzky et al., 2007]. The simplest discretization that can also be defined for graphs is a simple averaging of neighboring values without any encoding of positions in space:

$$\Delta f(v_i) = \frac{1}{\deg(v_i)} \sum_{v_j \in \mathcal{N}(v_i)} \left( f(v_j) - f(v_i) \right).$$

The most widely used discretization of the Laplace-Beltrami operator is the *cotangent Laplacian*, which can be derived in multiple different ways. It shows up among other cases when minimizing the *Dirichlet energy* over a triangle mesh in Pinkall and Polthier [1993] and during the computation of the *curvature normal* in Desbrun et al. [1999]. The cotangent Laplacian of a function $f : V \to \mathbb{R}^n$ where $V$ is the set of vertices of a triangle mesh S is defined by

$$\Delta f(v_i) = w_i \sum_{v_j \in \mathcal{N}(v_i)} w_{ij} \left( f(v_j) - f(v_i) \right). \tag{1.2}$$

The weights for vertices $w_i$ and edges $w_{ij}$ are defined as

$$w_i = \frac{1}{A_i}, \quad w_{ij} = \frac{1}{2}(\cot \alpha_{ij} + \cot \beta_{ij}), \tag{1.3}$$

where $A_i$ is the mixed Voronoi area of $v_i$ and $\alpha_{ij}$ and $\beta_{ij}$ are the interior angles opposite to edge $e_{ij}$ in the two triangles adjacent to $e_{ij}$. The cotangent Laplacian can be written in a matrix form as

$$\begin{pmatrix} \Delta f(v_1) \\ \vdots \\ \Delta f(v_n) \end{pmatrix} = \boldsymbol{L} \begin{pmatrix} f(v_1) \\ \vdots \\ f(v_n) \end{pmatrix}.$$

# 2. Related works

### Subdivision surfaces

We have already described how subdivision surfaces generally work in the introduction. The most popular subdivision scheme for triangle meshes is *Loop subdivision* [Loop, 1987]. A known algorithm that is also interpolatory is *butterfly subdivision* [Dyn et al., 1990]. Both schemes compute the new position of a vertex $v$ by weighted averaging of vertex positions in the proximity of $v$. Depending on the mesh topology around $v$ different vertices and weights are used for the averaging. These methods have linear time complexity in terms of the total number of vertices of the subdivided mesh. Loop subdivision gives generally better results but is not suitable for our task as it also recomputes the positions of the original vertices of the mesh. Butterfly subdivision is interpolatory and therefore solves exactly the problem of this thesis. Its drawback is that, in some cases, it produces undesirable oscillations of the resulting surface. It might be possible to add feature-retention to the algorithm in a similar way we add feature-retention to our approach. In the evaluation part of the thesis, we use modified butterfly subdivision [Zorin et al., 1996] to compare algorithm results on an organic mesh without sharp features.

### Curvature flow

This popular mesh smoothing approach was introduced in an article by Desbrun et al. [1999]. The idea is to move the vertices of the mesh in a direction that minimizes the mean curvature over the surface. This approach results in similar equations as the energy minimization method we employ in our own approach. The energy minimization method is in fact equivalent to the result of curvature flow if the flow would last for an infinite amount of time [Desbrun et al., 1999]. If given no constraints on the positions of vertices, this method slowly shrinks the entire mesh. The volume shrinkage is dealt with by rescaling the entire mesh uniformly after each iteration of curvature flow by the amount of overall volume that was lost.

### Neural subdivision

A recent approach to interpolatory subdivision was proposed by Liu et al. [2020]. In the introduction, we already described how to evaluate a subdivision scheme over a mesh dataset. This approach computes a loss function between an original fine and a subdivided coarse mesh and uses it to train a neural network with two dense layers and generally a very low number of weights. The network predicts the position of a vertex added during a subdivision surface scheme from the positions of already present close vertices—in this way, it is similar to traditional subdivision schemes. In the corresponding article, the network is trained only on one fine triangle mesh, which is randomly collapsed into multiple coarse meshes. The results are good, and the method is capable of feature retention when trained on a model with sharp features. It is not shown, however, what the method is capable of when trained over a large dataset.

After the article on neural subdivision was released, a new type of architecture for triangle mesh neural networks was described by Hu et al. [2022]. The architecture applies the concept of *convolutional neural networks* to triangle meshes. By convolution, the network passes information from local triangle neighborhoods to increasingly larger areas of the mesh. A trained network of this type can be used for mesh classification or segmentation and achieves state-of-the-art performance on those tasks. If this architecture was used for the task of interpolatory subdivision, it might achieve very good results. Subdivision by a neural network is also more efficient for larger meshes than, for example, an energy minimization approach because it scales linearly with the number of vertices of the mesh.

### Polynomial patches

A different way to solve the problem of this thesis is to fit all triangles with polynomial patches where two patches meeting at an edge have some degree of geometric continuity at their intersection. Then a subdivision scheme might be constructed that computes its vertices to be on those patches. The patches interpolate the vertices of the original mesh.

An easy-to-implement version of this can be seen in an article by Nagata [2005]. The method is able to preserve sharp edges and vertices. The patches, which are quadratic polynomials, however, meet at the edges at only $G^0$ continuity.

A method using *quartic triangular Bézier patches* and working with feature-retention is described in an article by Su and Senthil Kumar [2005]. It builds up on the work of Owen et al. [2002] and Walton and Meek [1996]. The resulting patches of the method have $G^1$ continuity at the edges. The article also covers how to compute the normals of vertices next to sharp features; these normals are then used for patch construction.

# 3. Our approach

## 3.1 High-level description

The core of our algorithm is solving a linear system of equations involving the cotangent Laplacian over a modified subdivided mesh. The result of solving the system are new positions for vertices added during the mesh subdivision. The vertices are placed such that a defined fairness energy over the mesh is minimized. This is also referred to in this text as solving the fairing equation. However, most of the complexity of our algorithm is in the preparation stage before the equation is solved. There are two important issues that need to be addressed in the preparation stage. First, the mesh from which we compute the cotangent Laplacian has to be topologically modified to prevent the smoothing of sharp features such as sharp edges and points. Second, the minimization of the fairness energy leads in some cases to a significant loss of volume of the resulting mesh, which should be prevented.

The algorithm can be summarized in the following steps:

1. **Detection of sharp edges** In the first step, we detect the sharp edges of the mesh. The mesh is later disconnected along the sharp edges. That way, the positions of vertices on different sides of the sharp edge will not affect each other when solving the fairing equation. To ensure the disconnection, we mark all sharp edges as *seams*. Seam is in this text a general term for an edge along which an original mesh is disconnected before solving the fairing equation.

2. **Detection of VGSs** The undesirable volume shrinkage occurs in triangle areas that resemble a part of the surface of a cylinder. We call these areas *vulnerable generalized triangle strips* (VGSs). The amount of volume shrinkage significantly decreases as the resembled cylinder becomes flatter. We therefore isolate the VGSs from the original triangle mesh and smooth out scaled down versions of them. We essentially apply a linear transformation that scales the vertices of the VGS down in the direction of the resembled cylinder height, fair the VGS using energy minimization, and then scale it back up. The boundary edges around a VGS separating it from the rest of the mesh are marked as seams.

3. **Creating seam paths** Seams consist of either VGS boundaries or sharp edges. If seams follow each other in a sequence in such a way that they form a curve, we denote them as a *seam path* (although a seam path can only consist of a single seam). The vertices of the seam path are fitted with polynomial curves, which we use later in the subdivision stage to compute the positions of new vertices on the seams. Figure 3.1 shows a model of a semicylinder with highlighted seams forming four seam paths.

4. **Subdivision** In this step, we subdivide each triangle uniformly into a given number of smaller triangles. The splitting along the seam paths is done by duplicating the vertices of the seams and connecting one set of vertices to

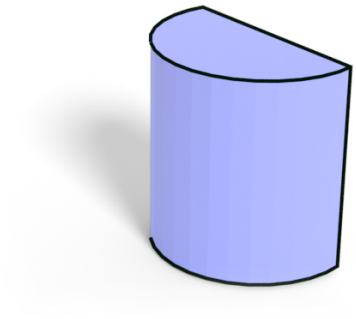


Figure 3.1: Semicylinder model with highlighted seams. All of the seams are sharp edges. In addition, the seams next to the round area of the model are VGS boundary edges.

triangles on one side of the seam path and the other set of vertices to triangles on the other side. Special care is taken when adding new vertices on seams. These vertices are fixed during the solving of the fairing equation, and therefore we need to compute the positions for them well, as the positions also appear in the final output of the algorithm. We interpolate the vertices of seam paths using cubic splines or quadratic curves respecting the surface normals. Because the vertices on seams are placed on curves, but the new vertices inside triangles are computed by linear interpolation, this creates a potentially high discontinuity between the vertices on the seam curves and the linearly interpolated inner vertices, which share an edge with them. A curve might, for example, run very close to or even in between some of the inner vertices. This could lead to wrongly set weights of the cotangent Laplacian. We therefore use only linear interpolation for vertices, from which we compute the cotangent Laplacian weights. Two positions for each vertex are stored:

1. Standard position that represents fixed points in the fairing equation. Can be potentially computed by interpolation of polynomial curves or patches.
2. Position for the cotangent Laplacian weights, which is always computed by linear interpolation.

We denote the points with changed positions, from which we compute the cotangent Laplacian weights, as *shadow vertices*.

5. **Computation of the cotangent Laplacian** What we compute in this step are the coefficients of a system of linear equations $\boldsymbol{Ax} = \boldsymbol{b}$, which contains the cotangent Laplacian weights and whose solution minimizes a defined fairness energy. The elements of this resulting matrix and vector can be obtained by traversing all paths along two or fewer edges of the input mesh, starting at a vertex that is free (is not being interpolated or is not a part of a seam). However, if a path would lead directly into a seam vertex, in order to have consistent cotangent Laplacian weights, some of the possible continuations of the path might lead *beyond the seam* to on the spot con-

structed final vertices. We denote these final vertices of the two edge path as *imaginary*. These imaginary vertices are placed on polynomial patches, which we create by interpolation from triangles neighboring the seam. In a way (although not exactly), by using the imaginary vertices, we define tangents to the resulting surface on the seam paths. An approximate visualization of what surface the imaginary vertices might form beyond the seams is in figure 3.2.

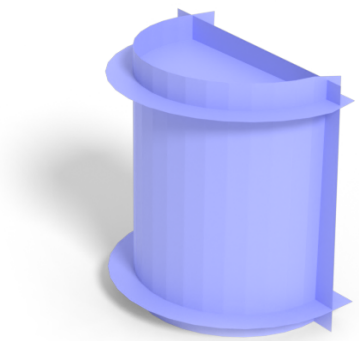


Figure 3.2: Semicylinder model with surface extensions.

Because in our approach, the position of an imaginary vertex is dependent on the first vertex of the path or how we got to the seam vertex, we compute imaginary vertices on the spot. This is necessary because imaginary vertices from all paths would not create a topologically consistent mesh that we could construct beforehand.

6. **Solving the fairing equation** In this part, we solve the system of linear equations from the previous step. As we will see, this is a sparse symmetric positive semi-definite system. The result of this step is a new position for each free vertex in the subdivided mesh. At the end of this step, we unite the duplicated seam vertices to return to the original topology. Figure 3.3 shows the result of the algorithm for the semicylinder model from figure 3.1.

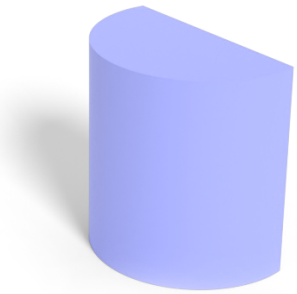


Figure 3.3: Semicylinder model as a result of our algorithm, where each edge is subdivided into eight smaller ones.

## 3.2 Fairing

### Thin-plate energy

Mesh fairing using the equation $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$, or more generally $\boldsymbol{L}^k\boldsymbol{x} = \boldsymbol{b}$, where $\boldsymbol{L}$ is the cotangent Laplacian, is a common paradigm in geometry processing. It can be used for smoothing out a region of the mesh, filling holes, or finding the smoothest deformations during mesh editing [Botsch and Sorkine, 2007]. The fairing equations can be derived by minimizing suitable energy functions over surfaces. *Thin-plate* energy that minimizes curvature is defined as

$$E_{\mathrm{TP}}(\boldsymbol{x}) = \iint_{\Omega} \kappa_{min} + \kappa_{max} \,\mathrm{d}u\mathrm{d}v,$$

where $\boldsymbol{x}$ is here a function $\boldsymbol{x}\colon \Omega \to \mathbb{R}^3$ that maps from a parameter space $\Omega \subseteq \mathbb{R}^2$ of a surface $S$ to $S$. Minimizing this energy, however, does not lead to solving a system of linear equations. In practice, a similar *linearized thin plate energy* that leads to a linear system and is therefore easier to compute is used instead:

$$\tilde{E}_{\mathrm{TP}}(\boldsymbol{x}) = \iint_{\Omega} \|\boldsymbol{x}_{uu}\|_2^2 + 2\|\boldsymbol{x}_{uv}\|_2^2 + \|\boldsymbol{x}_{vv}\|_2^2 \,\mathrm{d}u\mathrm{d}v.$$

The minimization of the above energy can be converted into a problem of solving the equation $\Delta_S^2\boldsymbol{x}(u,v) = 0$ over $\Omega$. For the problem to be well defined, it is necessary to introduce boundary constraints—typically in the form of positions and normals for the points over the boundary $\partial\Omega$. A discretization of the above equation leads to a linear system $\boldsymbol{L}^2\boldsymbol{x} = \boldsymbol{0}$ [Botsch et al., 2010], which is a cotangent Laplacian applied to another cotangent Laplacian that is applied to a function that returns the position of a vertex at each vertex. Using equations 1.2 and 1.3, we get the following:

$$\begin{aligned}\Delta^2\boldsymbol{x}_i &= w_i \sum_{j\in\mathcal{N}(v_i)} w_{ij}(\Delta\boldsymbol{x}_j - \Delta\boldsymbol{x}_i) \\ &= w_i \sum_{j\in\mathcal{N}(v_i)} w_{ij}\left(w_j \sum_{k\in\mathcal{N}(j)} w_{jk}(\boldsymbol{x}_k - \boldsymbol{x}_j) - w_i \sum_{k\in\mathcal{N}(v_i)} w_{jk}(\boldsymbol{x}_k - \boldsymbol{x}_i)\right),\end{aligned}$$

where $\boldsymbol{x}_i$ is the position of a vertex $v_i$. The equation $\boldsymbol{L}^2\boldsymbol{x} = 0$ is equivalent to $\Delta^2\boldsymbol{x}_i = 0$ for all vertices $v_i$. Defining boundary constraints for the discrete setting can be done by fixing the positions of certain vertices. The constrained discrete fairing problem is given by a system of linear equations

$$\boldsymbol{0} = \sum_{j\in\mathcal{N}(v_i)} w_{ij}\left(w_j \sum_{k\in\mathcal{N}(j)} w_{jk}(\boldsymbol{x}_k - \boldsymbol{x}_j) - w_i \sum_{k\in\mathcal{N}(v_i)} w_{jk}(\boldsymbol{x}_k - \boldsymbol{x}_i)\right) \tag{3.1}$$

for all $i$, where $v_i$ is a vertex that is not fixed. We denote the set of all free vertices as $V'$. The variables in the system are the positions $\boldsymbol{x}_i$ of all vertices $v_i \in V'$. The solution of the system are the positions of vertices $v_i \in V'$ that minimize the discrete linearized thin plate energy over the triangle mesh.

A different type of fairness energy is the *membrane energy*

$$E_{\mathrm{M}}(\boldsymbol{x}) = \iint_{\Omega} \sqrt{\det(\mathbf{I})} \,\mathrm{d}u\mathrm{d}v,$$

which minimizes surface area. The expression $\det(\cdot)$ denotes a determinant of a given matrix. The discretization of the linearized form of $E_{\mathrm{M}}$ with added boundary constraints leads to the system $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$. The system for thin-plate energy can be written in a similar form $\boldsymbol{L}^2\boldsymbol{x} = \boldsymbol{b}$. Higher order equations in the general form $\boldsymbol{L}^k\boldsymbol{x} = \boldsymbol{b}$ are also used—the equation $\boldsymbol{L}^3\boldsymbol{x} = \boldsymbol{b}$ corresponds to the minimization of *minimum variational energy* [Botsch et al., 2010]. The number $k$ corresponds to a smoothness $C^k$ on the boundary of the optimized continuous surface. Figure 3.4 shows the comparisons between membrane and thin-plate energy optimization.

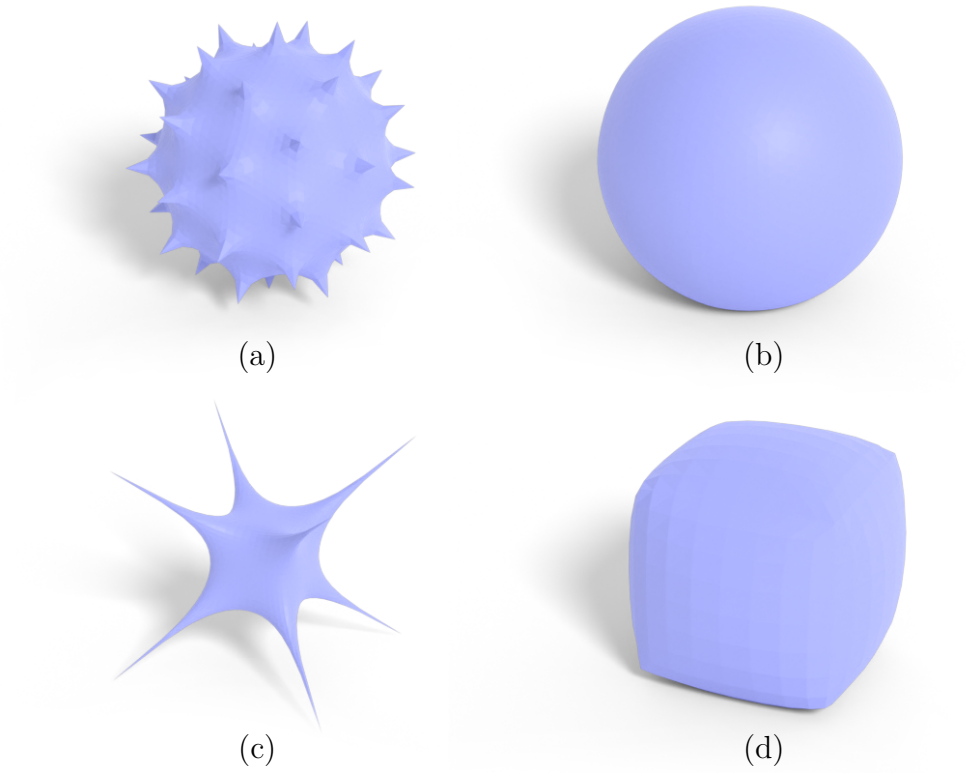



Figure 3.4: Comparison of membrane (a, c) and thin-plate (b, d) energy optimization. The optimization was performed on uniformly subdivided sphere (a, b) and cube (c, d) meshes. The positions of the original vertices that were present in the meshes before subdivision were fixed during the optimization.

In our algorithm, we employ thin-plate energy optimization and interpolate polygonal boundary vertices as well as single isolated points corresponding to the original vertices of the mesh—this is a different process from the applications mentioned in the beginning of this section that *do not interpolate* single points. We have, however, found a mention of point constraints added to thin-plate energy optimization in Jacobson et al. [2010].

Note that we do not solve the equation $\boldsymbol{L}^2\boldsymbol{x} = \boldsymbol{b}$ directly but instead solve an equivalent linear system with some rows and columns corresponding to fixed vertices removed.

### Coefficients of the linear system

The computation of the linear system from equation 3.1 can be done in a path traversal way over the mesh. To compute the coefficients corresponding to the equation associated with a vertex $v_i \in V'$, we do a depth-first search starting at $v_i$. At the beginning of the search, we assign zero to all coefficients corresponding to the equation. During the search, we update a variable $t$, which is the multiplication of all weights encountered on the path from $v_i$. A path is followed in the following steps:

1. In the first step, we start at $v_i$ with $t = 1$. From here, we can either move to $v_j \in \mathcal{N}(v_i)$ and multiply $t$ by $w_{ij}$, or we can stay at $v_i$ and multiply $t$ by the negative sum of all weights $w_{ij}$ for $j \in \mathcal{N}(v_i)$.

2. In the second step, we repeat the same process as in the first step: we either stay at the current vertex or go to a neighboring vertex. During this, we again multiply $t$ by the corresponding weights computed in the same way as in step 1. In addition, we multiply $t$ by the weight $w_j$ of the initial vertex of the second step.

3. In the final step, we are at a vertex $v_k$. If $v_k \in V'$, we add $t$ to the current value of the coefficient corresponding to $v_k$ in the equation for $v_i$. If $v_k \notin V'$, we add $-t$ to the coefficient $b$, which does not correspond to any variable.

If we traverse all possible paths in the depth first search, we get the final coefficients

$$a_1\boldsymbol{x}_1 + \cdots + a_n\boldsymbol{x}_n = b,$$

where $\boldsymbol{x}_1, \ldots \boldsymbol{x}_n$ are the positions of vertices at most two edges far from $v_i$ that belong to $V'$. This is how we compute the coefficients of the linear system for each row in our implementation. The resulting matrix of the system is sparse because in each row the amount of non-zero elements is equal to the number of points $v_k$ reached from a point $v_i$ by a path that contains at most one other vertex between them (we assume the number of reachable points $v_k$ from a given vertex $v_i$ is on average a constant number).

To show that the resulting matrix from the system is symmetric and positive semi-definite, we define matrix $\boldsymbol{D}$ as a diagonal matrix with weights $w_i$ on the diagonal and a matrix $\boldsymbol{L}_s$ as

$$(\boldsymbol{L}_s)_{ij} = \begin{cases} -\sum_{v_k \in \mathcal{N}(v_i)} w_{ik}, & i = j, \\ w_{ij}, & v_j \in \mathcal{N}(v_i), \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

Furthermore, we denote $\boldsymbol{L}'_s$ as the matrix $\boldsymbol{L}_s$ with all rows corresponding to the fixed vertices $v_i \notin V'$ removed. Note that columns of the matrix correspond to all original vertices, free vertices from $V'$, and vertices on seams. Imaginary vertices are not part of the matrix $\boldsymbol{L}'_s$, as we will see in section 3.5.

The system of linear equations from equation 3.1 is equivalent to

$$\boldsymbol{L}'_s\boldsymbol{D}(\boldsymbol{L}'_s)^T\boldsymbol{x} = \boldsymbol{b}. \tag{3.3}$$

The weights in the previous equation should correspond to the weights encountered during the depth first search and also to the weights in equation 3.1. Because values of mixed Voronoi areas $w_i$ are always positive, the equation 3.2 can be written as

$$\left(\boldsymbol{L}'_s\sqrt{\boldsymbol{D}}\right)\left(\boldsymbol{L}'_s\sqrt{\boldsymbol{D}}\right)^T\boldsymbol{x} = \boldsymbol{b}.$$

The system from equation 3.1 contains a matrix of the form $\boldsymbol{A}\boldsymbol{A}^T$; it can be shown using basic linear algebra that the system is symmetric and positive semi-definite. A proof of negative definiteness that applies to the matrix $\boldsymbol{L}$ can be found in Pinkall and Polthier [1993] and is a part of the derivation of the cotangent Laplacian. The book Botsch et al. [2010] references this proof and shows that the matrix denoted in the book as $\boldsymbol{M}(\boldsymbol{D}\boldsymbol{M})^2$ and a version of the same matrix with rows and columns corresponding to fixed vertices removed is positive definite. The matrix $\boldsymbol{M}(\boldsymbol{D}\boldsymbol{M})^2$ with rows and columns removed is equivalent to the matrix of the linear system from equation 3.3. From this assumption, the linear system corresponding to equation 3.3 is positive definite.

In our algorithm, we construct imaginary vertices on the spot at the ends of certain weight computation paths. The imaginary vertices are not stored anywhere; when tracing a path, we compute the positions of the imaginary vertices, use them to determine cotangent Laplacian weights, and then discard them. These vertices, taken together, do not create a consistent mesh topology. Because of that, we cannot apply the previously stated proofs of positive definiteness to our algorithm. In section 3.5, we suggest how to modify our algorithm to lead to a positive definite system at the cost of having to manually compute the positions of some previously free vertices.

Note that our implementation also contains the energy solutions for $\boldsymbol{L}\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{L}^3\boldsymbol{x} = \boldsymbol{b}$. These correspond to edge paths of lengths at most one and at most three respectively and also lead to positive definite linear systems.

### Solving the linear system

The book by Botsch et al. [2010] discusses approaches to solving sparse symmetric positive definite systems and also compares them using the task of computing $\boldsymbol{L}^2\boldsymbol{x} = \boldsymbol{b}$ for larger triangle meshes. The two best approaches in those comparisons are *multigid iterative solvers* and *sparse direct Cholesky solver*. The Cholesky solver factorizes a matrix $\boldsymbol{A}$ from the equation $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ into $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$ where $\boldsymbol{L}$ is a lower triangular matrix. For a triangle mesh the same equation needs to be solved three times for each coordinate $x, y, z$ of $\mathbb{R}^3$. The factorization is done only once because in our case only the right hand side $\boldsymbol{b}$ changes for each coordinate, not the matrix. The time complexity of the Cholesky solver is in the worse case $O(n^3)$ for the factorization and $O(n^2)$ for the subsequent substitution. However, if the input matrix has special properties such as low *bandwidth* which can be found in triangle meshes [Botsch et al., 2010], the complexity of the factorization decreases.

Cholesky decomposition can be applied only to positive definite systems. Our algorithm would need to be modified to result into a positive definite system (see section 3.5) or it results into a positive definite system if feature-preservation and volume shrinkage prevention are turned off. In our implementation we use a similar approach instead called *LDLT decomposition* that decomposes matrix $\boldsymbol{A}$

into $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ where $\boldsymbol{L}$ is a lower triangular matrix and $\boldsymbol{D}$ is a diagonal matrix. As opposed to Cholesky decomposition, LDLT decomposition can also decompose symmetric positive semi-definite matrices which is the case with our algorithm.

In our implementation, we use the `SimplicialLDLT` class from the Eigen library (`https://gitlab.com/libeigen/eigen`) for the LDLT decomposition.

## 3.3 Feature detection and seam interpolation

The thin-plate energy minimization described in the previous section, in combination with a uniform subdivision of the triangles of the input mesh, can already be used as a functioning algorithm for interpolatory mesh fairing. However, the sharp features of the input mesh will get lost in the energy minimization process. We count as sharp features all areas that are not $G^1$ continuous, meaning that the tangent plane to the surface at those points is not continuous. On a triangle mesh, we detect two kinds of discontinuities. Sharp edges of dimension 1 and *sharp fans* of dimension 0.

### Detection of sharp edges

Our detection of sharp edges uses methods mentioned in the article by Hubeli et al. [2000]. There are two angle parameters: $\alpha$ and $\beta$, where $\alpha \leq \beta$. For each edge in the triangle mesh, we compute its dihedral angle and classify the edges into three groups:

1. Edges with dihedral angle $\varphi \leq \alpha$. These edges are automatically considered not sharp.

2. Edges with dihedral angle $\varphi \geq \beta$. These edges are automatically sharp.

3. Edges with dihedral angle $\varphi \in (\alpha, \beta)$. If an edge $e$ from this group has an adjacent edge that belongs to the second group of automatically sharp edges, it is also considered sharp. Otherwise, it is considered not sharp.

This approach to sharp edge classification is called *hysteresis thresholding* and is the method we use in our implementation.

A potentially better method from the article by Hubeli et al. [2000], that we did not implement, is *angle between best fit polynomials*, which measures an angle between two polynomials that interpolate points on the edges of the triangle mesh close to the classified edge. The polynomials are approximating curves on the surface of the mesh perpendicular to the measured edge. One polynomial is approximating a curve starting in the middle of the measured edge and going in one direction perpendicular to the edge. The other polynomial starts at the same point and interpolates points on the opposite side of the measured edge. The method computes a similar value as the dihedral angle but takes into account a bigger neighborhood than the two adjacent triangles of the dihedral angle.

### Composition of seams into seam paths

We denote as a *seam path* a sequence of vertices $v_1, \ldots, v_n$ of the triangle mesh where the edges formed by neighboring vertices in the sequence are seams. By seams, we mean sharp edges and boundary edges of VGSs. In addition, in a seam path, the angle between the directions of two neighboring vectors must be less than an angle parameter $\gamma$. A seam path is, in a sense, a smooth curve that separates two areas of the triangle mesh. We allow the repetition of vertices in $v_1, \ldots, v_n$ but prohibit the repetition of the same pairs of successive vertices, even in reversed pair order. If $v_1 = v_n$ the seam path forms a loop.

To determine unique seam paths for each triangle mesh, a conflict needs to be resolved where a seam can potentially have two following seams whose angle with the original seam is less than $\gamma$. We resolve this by connecting two seams $e_1$ and $e_2$ at a vertex $v$ that have an angle $\theta < \gamma$ between their edge directions only if the angle between $e_1$ and any other seam incident to $v$ different from $e_2$ is greater than $\theta$ and the angle between $e_2$ and any other seam incident to $v$ different from $e_1$ is greater than $\theta$. This rule uniquely determines the seam connections in a given vertex and, in turn, for the whole mesh. It also should not be possible to get a seam path where the same pair of successive vertices is repeated.

Because the seam paths are unique for each triangle mesh and for each vertex we can determine what two edges are on the same path, a seam path can easily be computed given one edge of the path by traversing the path in both directions. In our implementation, we iterate over all seams and, for each seam, trace the seam path if the edge is not already part of an established seam path.

### Interpolation of seam paths

During the solving of the fairing equation, the new points on the seam paths created by subdivision will be fixed. For the computation of the positions of the points, we employ *cubic splines* defined in $\mathbb{R}^3$. A seam path can be divided into segments, where each segment corresponds to one original seam or two successive vertices of the seam path sequence. In a segment going from $v_i$ to $v_{i+1}$, the cubic spline is represented by a polynomial

$$Y_i(t) = \boldsymbol{a}_i + \boldsymbol{b}_i t + \boldsymbol{c}_i t^2 + \boldsymbol{d}_i t^3,$$

where $t \in [0, l_i]$, $\boldsymbol{a}_i, \boldsymbol{b}_i, \boldsymbol{c}_i, \boldsymbol{d}_i \in \mathbb{R}^3$ are the coefficients of the polynomial $Y_i$, and $l_i$ corresponds to the assigned length of $Y_i$. The polynomial $Y_i$ interpolates the two vertices— $Y_i(0)$ should be the position of $v_i$ and $Y_i(l_i)$ the position of $v_{i+1}$. In addition, for successive polynomials $Y_i$ and $Y_{i+1}$, we require that

$$Y_i'(l_i) = Y_{i+1}'(0) \quad \text{and} \quad Y_i''(l_i) = Y_{i+1}''(0),$$

so that the whole spline has continuous first and second derivatives. We solve a sparse linear system to get the coefficients separately for each coordinate $x, y, z$ of $\mathbb{R}^3$. Given a seam path $v_1, \ldots v_n$, the total number of polynomials for the spline is $n-1$ and the total number of coefficients is $4(n-1)$. There are $2(n-1)$ equations for the positions of the endpoints of the polynomials and $2(n-1)-2$ equations for the first and second derivatives. Two more equations are needed to get a linear system with a square matrix. The two equations are commonly

added in the form of boundary conditions for the first and last polynomials; often used is the *natural* boundary condition that sets the second derivatives at the two endpoints of the spline to zero. In our implementation, we use first derivative equality with quadratic curves over single edges, which we describe later in this section. An exception to the boundary conditions are loops where the first and second derivative equality equations can be defined at $v_0 = v_n$ instead of the boundary conditions.

What we have not addressed yet are the lengths $l_i$ assigned to the polynomials. A common way is to set them to be proportional to the lengths of the original seam edges. A different approach called the *centripetal model*, proposed in an article by Lee [1989], produces in general results that more closely follow the interpolated vertices. Instead of setting the length $l_i$ to be $\|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\|_2$, where $\boldsymbol{x}_i, \boldsymbol{x}_{i+1}$ are the positions of $v_i, v_{i+1}$, respectively, we set it to

$$\|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i\|_2^a.$$

The parameter $a$ is recommended to be $1/2$, which corresponds to the square roots of edge lengths.

In our implementation we use the previously described centripetal model with $a = 1/2$. In addition, we solve the system of linear equations for a cubic spline using *sparse LU decomposition*.

### Computation of normals

In this section, we cover the computation of the normals of a vertex. Vertex normals can easily be computed using angle-weighted averaging introduced in section 1.2.1. However, if there are sharp edges adjacent to a vertex, the vertex does not have a defined tangent plane, and therefore it is not clear how the normal should look. Instead of computing a normal for an entire vertex, we compute a normal with respect to a vertex and an edge adjacent to it.

If we take a closed triangle fan around a vertex $v$, the fan can be partitioned into open triangle fans by splitting the closed fan along sharp edges. Such open fans contain sharp edges only at their boundaries. We denote the partitioned fans as *smooth fans* of $v$. In most cases, smooth fans have defined normals, although the computation of a normal corresponding to an edge of a smooth fan is also not straightforward, and we describe various cases of what steps need to be taken to compute it in the text below. It is important to add that what we compute is actually a normal at a vertex from the direction of an adjacent edge—the smooth fans only in some cases make the computation easier. Also, when the adjacent edge is a sharp edge, there are actually two different normals from both sides of the edge, which we compute by first making the edge associated with one smooth fan next to the sharp edge and then the other. Here are the cases we need to consider for a vertex $v$ and an adjacent edge $e$:

1. The vertex $v$ does not have any sharp edges adjacent to it, or the splines for the seam paths have not been computed yet, and we therefore cannot access their coefficients—this happens when computing the splines themselves. In this case, we take an angle-weighted normal of the triangles of the smooth fan associated with $e$. Angle-weighted normal averaging is discussed in section 1.2.1.

2. The smooth fan associated with $e$ is bounded by two sharp edges from the same seam path going through $v$, or there is only one sharp edge containing $v$ (splines are computed). In this case, we want the normal to be consistent with the tangent of the spline of the seam path of the two sharp edges or one edge at $v$. The result of this case is the angle-averaged normal of the smooth fan projected into a plane defined by the spline tangent at $v$.

3. There are two sharp edges from distinct seam paths with computed splines that are on the boundaries of a smooth fan associated with $e$. Then we take as normal the cross-product of the two splines' tangents at $v$. The two tangents are enough to define a normal, and we therefore do not take into account the normals of the triangles of the smooth fan.

4. The smooth fan associated with $e$ does not have a defined tangent plane. This is a case that might happen, for example, on the tip of a cone. There are no sharp edges next to the tip, but the tip is not $G^1$ continuous. We denote these smooth fans as *sharp fans* and cover how to detect them in the next section. This case has precedence over the previous cases; if a sharp fan is encountered, the normal is handled as a sharp fan normal. If $e$ is a sharp edge, the computed normal is the normal of one of the neighboring triangles, depending on from which side we compute the normal. If $e$ is not a sharp edge, the computed normal is the average of the normals of the two neighboring triangles of $e$.

### Sharp fans

Sharp fans cover the cases when a surface is not smooth at a single point. As these points generally resemble partial or whole sharp tips, we use discrete Gaussian curvature to identify them—concretely integrated Gaussian curvature, as we want our detection of sharp fans to be independent of the local scale of the mesh. A higher positive Gaussian curvature should correspond to the sharp tips, for a parameter $\varphi$ we therefore state that a smooth fan of a vertex $v$ is a sharp fan if it satisfies

$$\tilde{K} = 2\pi - \frac{1}{k}\sum_i \theta_i > \varphi, \tag{3.4}$$

where $i$ iterates over all interior angles $\theta_i$ of the smooth fan. Because the smooth fan might cover only a part of the triangles adjacent to $v$, we want to somehow scale the values of the sum to an amount corresponding to the values of similar triangles around the whole closed fan. The factor $k$ states how much of the vertex neighborhood the smooth fan covers. In our implementation, we first compute a simple angle-weighted normal $\boldsymbol{n}_s$ of the vector $v$. Then we project all triangles of the smooth fan into a plane defined by $\boldsymbol{n}_s$ and compute the total sum of the interior angles corresponding to $v$ of the projected triangles. This sum divided by $2\pi$ is the factor $k$.

### Quadratic interpolation of edges

For the boundary conditions of non-loop splines, we use first derivative equality with quadratic curves constructed over the first and last edges of the corresponding seam path. During the construction of these quadratic curves, we

*take into account the normals* of the two vertices of an edge—for a seam path, these are either $v_1, v_2$ or $v_{n-1}, v_n$. The first derivative equality should be satisfied at points $v_1$ and $v_n$. The quadratic curves over single edges we now introduce are also utilized during the polynomial patch interpolation described later. The quadratic curves allow for an easy way, implementation-wise, to make the ends of the cubic splines more in line with the neighboring surface. If a seam path has an end vertex where there are no other sharp features apart from the line itself, we want the path to, in a way, smoothly transition into the neighboring surface. This might be defined as wanting to have the spline tangent have the same direction as an existing directional derivative of the neighboring surface at the spline endpoint. To compare, a natural boundary condition might, in some cases, lead to a greater deviation of the spline tangent from the neighboring surface.

The method we use for quadratic curve interpolation was introduced in an article by Nagata [2005]. A quadratic curve between vertices $v_1$ and $v_2$ has the form

$$P(t) = \boldsymbol{a} + \boldsymbol{b}t + \boldsymbol{c}t^2,$$

where $t \in [0, 1]$ and $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{R}^3$ are the coefficients of the polynomial $P$. According to the article, the polynomial $P$ interpolates the two vertices and, in addition, is trying to be perpendicular to two arrays of normals $\boldsymbol{n}_1, \cdots, \boldsymbol{n}_k$ and $\boldsymbol{n}'_1, \cdots, \boldsymbol{n}'_l$, which correspond to the vertices $v_1$ and $v_2$, respectively. This leads to a system of equations where, in addition to the interpolation of two points, there are equations for each normal. In the equation for a normal, the dot product of the normal and the tangent of the curve at a corresponding point is equal to zero. With two or fewer normals in total, the system is underdetermined and the method finds a solution with the minimal value for $|\boldsymbol{c}|$. For more normals than three, the system is overdetermined and the method finds a solution with the minimal value for $|\boldsymbol{c}|$ among all least-square solutions. In contrast to the cubic splines solution, this method solves the problem as one linear system and not three systems for each coordinate of $\mathbb{R}^3$. The method is also explicit, so no calling of a matrix solving function is needed. We do not describe the steps of the method here, but they can be viewed in the original article by Nagata [2005].

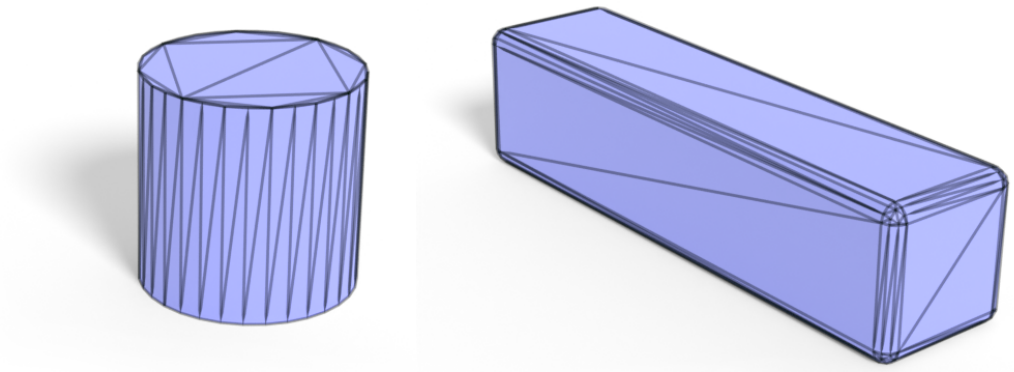
With the quadratic curves described, we can now return to the problem of spline boundary conditions of seam paths. Given a seam edge $e$ at the end of a seam path, we compute normals in both of its vertices in the direction of $e$, as described previously in this section in a paragraph about normals. The results are two different normals for each vertex if $e$ is a sharp edge and one normal for each vertex if $e$ is a VGS boundary edge. These normals are put as input parameters to the quadratic curve construction over the edge $e$. The first derivative of the curve is computed from the coefficients of the resulting polynomial $P$. Because the curve is parametrized with $t \in [0, 1]$, we need to rescale the first derivative appropriately so it corresponds to the first derivative of the same but reparametrized curve with $t \in [0, l]$ where $l$ is the centripetal length of the corresponding spline segment upon which we want to impose the tangent equality condition.

In addition, if a seam path has only two points, we use the coefficients of the quadratic curve constructed over the single edge of the path directly as the spline coefficients of the single spline segment. In this case, we entirely skip the spline equation solving.

## 3.4 Prevention of volume shrinkage

### Volume shrinkage analysis

In some cases, the minimization of thin-plate energy over the triangle mesh leads to a significant loss of volume in the resulting smoothed-out mesh. In general, the reduction seems to be happening in areas that resemble the curved surface of a cylinder or a cone—larger curved areas with no vertices to interpolate in the middle of them (see parts of shapes in figure 3.5). If we run our algorithm on meshes with such areas without any volume shrinkage prevention, we get a mesh with vertices shifted inward and reduced volume (see figures 3.6 and 3.7). The amount of surface distortion depends strongly on the *height* of the resembled cylinder, as can be seen in figure 3.7.



(a) (b)

Figure 3.5: Triangle meshes with highlighted edges—a cylinder (a) and a block with rounded edges (b).

In the following text, we compute how a cylinder is distorted in the continuous case. This does not cover the more general cases where a mesh resembles a part of a cylinder only vaguely, but it might, at least for the most basic case, give some insight into the problem. We represent the curved surface of a *smoothed out*

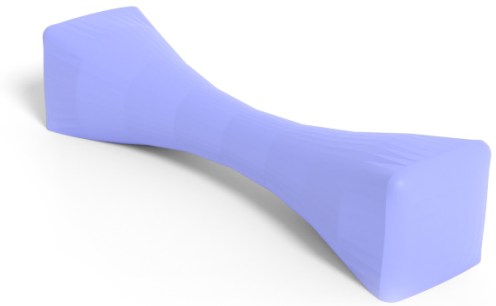


Figure 3.6: A triangle mesh from figure 3.5 (b) as a result of our algorithm with no volume shrinkage prevention.

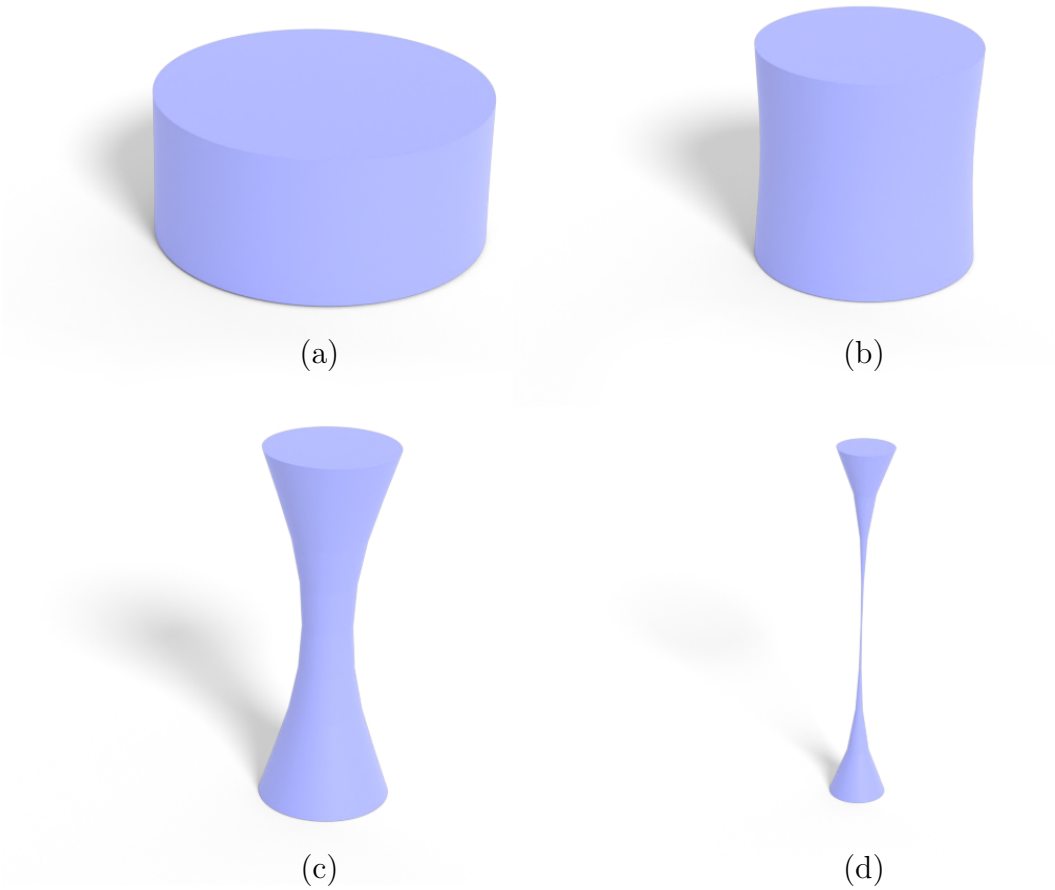



Figure 3.7: Triangle meshes from figure 3.5 (a) with different scale applied to them before being processed by our algorithm without volume shrinkage prevention.

cylinder by a function

$$g(x,y) = \begin{pmatrix} f(x)\cos y \\ f(x)\sin y \\ x \end{pmatrix}, \tag{3.5}$$

where $f(x)$ is a function describing the radius of the cylinder at height $x$ and $y \in [0, 2\pi]$. An assumption that we make is that we can actually use the function $f(x)$ to describe the cylinder after energy minimization. In the next step, we compute the squared Laplacian (bilaplacian) of the function $g$ (note that this is not rigorous as we do not use the Laplace-Beltrami operator for manifolds but only the standard Laplacian in $\mathbb{R}^3$). Function $g$ is vector valued, so the result is a vector of bilaplacians for each coordinate of $g$:

$$\begin{aligned} \Delta^2 g(x,y) &= \Delta\left(\frac{\partial^2 g}{\partial x^2}(x,y) + \frac{\partial^2 g}{\partial y^2}(x,y)\right) \\ &= \frac{\partial^4 g}{\partial x^4}(x,y) + 2\frac{\partial^4 g}{\partial x^2 \partial y^2}(x,y) + \frac{\partial^4 g}{\partial y^4}(x,y) \\ &= \begin{pmatrix} \left(f^{(4)}(x) - 2f^{(2)}(x) + f(x)\right)\cos y \\ \left(f^{(4)}(x) - 2f^{(2)}(x) + f(x)\right)\sin y \\ 0 \end{pmatrix}. \end{aligned}$$

We search for a function $f(x)$ for which the bilaplacian is zero over the whole parameter space of $g(x,y)$, i.e., the surface defined by $f(x)$ has minimal thin-plate energy. For arbitrary $x$, there is always a $y$ (for example $y = \pi/3$), where $\cos y$ is not zero. Because of that, it must be true that

$$f^{(4)}(x) - 2f^{(2)}(x) + f(x) = 0 \tag{3.6}$$

for every $x \in [0, t]$, where $t$ is the height of the cylinder. For simplicity, we set the radius of the cylinder to one but keep the height $t$ of the cylinder as a variable. With appropriate scaling, the solution should apply to arbitrary cylinders. The formula for the function $f$ can be computed as a result of a ordinary differential equation defined by equation 3.6 with added constraints

$$f(0) = 1, \quad f(t) = 1, \quad f'(0) = 0, \quad f'(t) = 0.$$

The first two equations represent that we interpolate the rings of the two bases of the cylinder. The last two equations state that the tangents near the bases are perpendicular to the bases. This simulates the fact that in our algorithm, we construct imaginary vertices in approximately tangential directions to the bases of the cylinder as a continuation of the mesh. We compute the solution of the differential equation using the following line in *Mathematica* [Wolfram, 2023]:

```
DSolve[{y[x] - 2 y''[x] + y''''[x] == 0,
y[0] == 1, y[t] == 1, y'[0] == 0, y'[t] == 0}, y[x], x]
```

The result is the following single solution:

$$f(x) = \frac{e^{-x}\left(te^{t+2x} + e^{t+2x} - e^t x + e^{2t}x - xe^{t+2x} + e^t t - e^t + e^{2t} - e^{2x} + e^{2x}x\right)}{2e^t t + e^{2t} - 1}$$

.

When plotting the function $f(x)$ for different values of $t$, we obtain similar results as the output of our algorithm with no volume shrinkage prevention for differently sized cylinders. This can be seen in figures 3.8, 3.9 and 3.10.

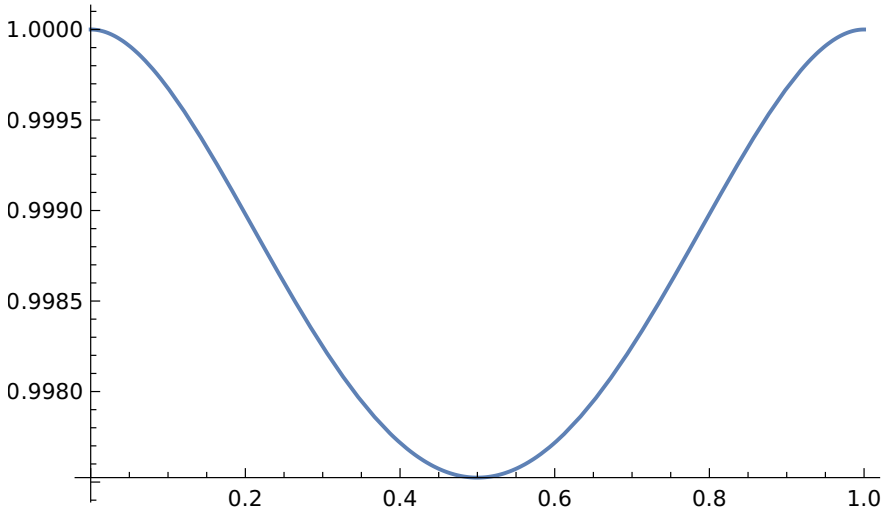



Figure 3.8: The values of $f(x)$ from equation 3.5 for $t = 1$.

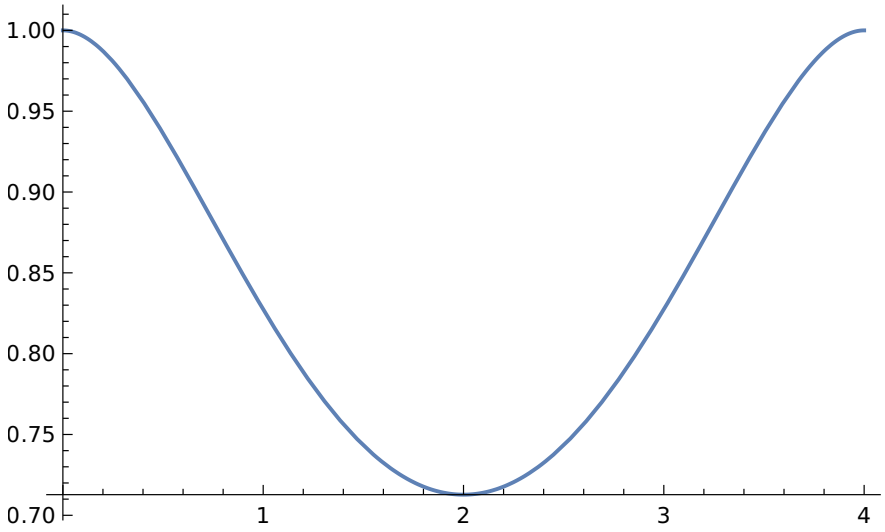



Figure 3.9: The values of $f(x)$ from equation 3.5 for $t = 4$.

### Preventing the volume shrinkage

Our solution to the shrinkage problem involves the detection of vulnerable generalized triangle strips (VGSs). A VGS represents a generalized triangle strip prone to volume shrinkage. To each VGS, we assign a unit vector $\boldsymbol{v}$, which approximately corresponds to the direction from a lower base to an upper base of a cylinder resembled by the VGS. Another value assigned to the VGS is a scalar $s \in (0, 1)$, which denotes the amount of shrinkage necessary to prevent noticeable volume shrinkage. From the values of $\boldsymbol{v}$ and $s$, a non-singular matrix $\boldsymbol{T}$

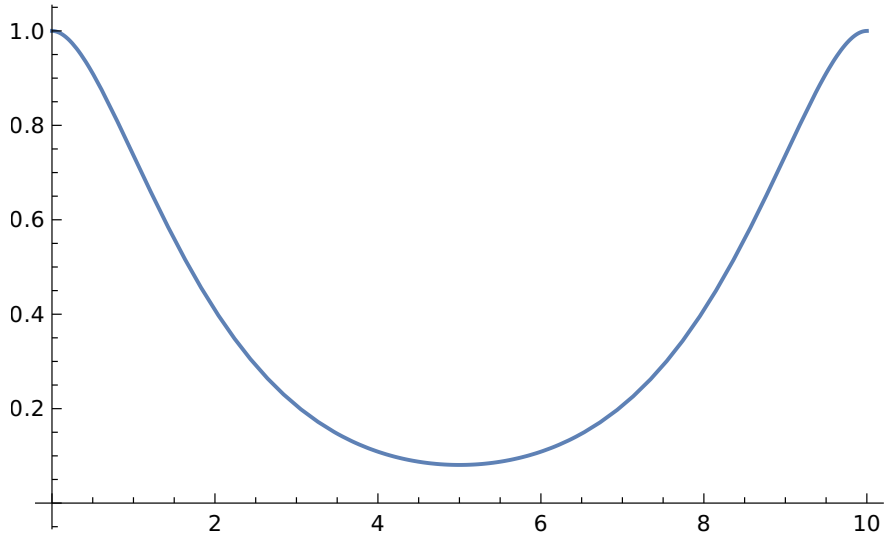



Figure 3.10: The values of $f(x)$ from equation 3.5 for $t = 10$.

is constructed, which scales space in the direction of $\boldsymbol{v}$ by a factor of $s$. The matrix $\boldsymbol{T}$ has the form

$$\boldsymbol{T} = \left(\boldsymbol{v}\boldsymbol{v}^T(s-1) + \boldsymbol{I}\right).$$

This holds because

$$\begin{aligned}\boldsymbol{T}\boldsymbol{x} &= \left(\boldsymbol{v}\boldsymbol{v}^T(s-1) + \boldsymbol{I}\right)\boldsymbol{x} \\ &= \left(s\boldsymbol{v}\boldsymbol{v}^T - \boldsymbol{v}\boldsymbol{v}^T + \boldsymbol{I}\right)\boldsymbol{x} \\ &= (-\boldsymbol{v}\boldsymbol{v} + \boldsymbol{I})\,\boldsymbol{x} + s\boldsymbol{v}\boldsymbol{v}^T\boldsymbol{x}; \end{aligned} \tag{3.7}$$

the expression $\boldsymbol{v}\boldsymbol{v}^T\boldsymbol{x}$ represents a projection of $\boldsymbol{x}$ into $\boldsymbol{v}$ and $s\boldsymbol{v}\boldsymbol{v}^T\boldsymbol{x}$ is a scaled-down version of it. The expression $(-\boldsymbol{v}\boldsymbol{v} + \boldsymbol{I}\,)\,\boldsymbol{x}$ is the component of $\boldsymbol{x}$ perpendicular to $\boldsymbol{v}$. The matrix $\boldsymbol{T}$ scales down only the component of $\boldsymbol{x}$ parallel to $\boldsymbol{v}$ and keeps the perpendicular component intact.

As we have stated in the high-level description of the algorithm, we isolate the VGSs from the rest of the mesh by marking the boundary edges of the VGSs as seams. Before thin-plate energy minimization, we transform all vertices of the VGS, including seam and imaginary vertices, using the matrix $\boldsymbol{T}$. After the energy minimization, all vertices are scaled back using the inverse of $\boldsymbol{T}$ and are subsequently connected back to the now smoothed out mesh.

### VGSs

In this part, we first show how to deterministically find potential VGSs of a triangle mesh. Then we describe how to process the individual potential VGSs to determine corresponding scale matrices $\boldsymbol{T}$. During the processing of the strip, we might find that a potential VGS is not actually a VGS, and in that case, we either discard it or divide it into two shorter potential VGSs that are processed subsequently.

A VGS is a sequence of neighboring triangles $t_0, \ldots, t_{n-1}$ that form a generalized triangle strip. Triangles $t_i$ and $t_{i+1}$ share an edge $e_{i+1}$, which we denote as

*interior edge of a VGS*. It is possible for a VGS to form a loop when triangles $t_0$ and $t_n$ share an edge—we mark the edge as $e_0$ (this is the case for a cylinder from figure 3.5). We denote as *potential VGSs* all maximal with respect to inclusion generalized triangle strips whose angles between successive interior edges $e_i$ and $e_{i+1}$ are lower than a given threshold $\gamma < 60°$. This should give us a sequence of thin neighboring triangles where there is a significant relative distance between the ends of edges $e_0, \ldots, e_n$ on one side of the VGS and the other. In addition, we add a rule that if a triangle has two interior angles with values less than $\gamma$, then only the two edges $e_i, e_{i+1}$ corresponding to a sharper of these two angles can be in the same potential VGS. This should be enough to deterministically evaluate all potential VGSs of a triangle mesh (considering that potential VGSs are maximal with respect to inclusion sequences of triangles). It is possible to evaluate for each triangle if two of its edges can be in the same potential VGS and which one these are. All potential VGSs can be computed by iteration over all triangles, where for each triangle we trace its potential VGS if it is not already a part of one created earlier. This approach is similar to the detection of seam paths described in section 3.3.

Having a VGS, it is necessary to determine its scale direction $\boldsymbol{v}$. For each triangle $t_i$, we denote as a base vector $\boldsymbol{b}_i$ the vector that has the same direction and magnitude as an edge of $t_i$ that is neither $e_i$ nor $e_{i+1}$. For a non-loop VGS and triangles $t_0$ and $t_{n-1}$, we simply take for each triangle the shorter of the two edges that are not $e_1$ or $e_{n-1}$. The base vectors correspond to the vectors of edges in the two bases of a cylinder that the VGS resembles. In a similar way, the scale direction $\boldsymbol{v}$ should somehow resemble the direction of the cylinder height—the direction from the lower base to the upper base of the cylinder that is perpendicular to both bases. The vector $\boldsymbol{v}$ can be the solution of a problem given by the following definition:

**Definition 11** (Optimal VGS scale direction)**.** *A unit vector $\boldsymbol{v}$ is an optimal scale direction of a VGS with n triangles if it satisfies*

$$\begin{aligned}\boldsymbol{v} &= \arg\min_{\boldsymbol{x}} \sum_{i=0}^{n-1} (\boldsymbol{b}_i \cdot \boldsymbol{x})^2 \\ &= \arg\min_{\boldsymbol{x}} \|\boldsymbol{B}\boldsymbol{x}\|_2,\end{aligned}$$

*where $\boldsymbol{x}$ a vector of unit length, $\boldsymbol{b}_0, \cdots \boldsymbol{b}_{n-1}$ are base vectors of the VGS that can have arbitrary values, and $\boldsymbol{B}$ is a matrix composed of the base vectors of the VGS as rows.*

Because the vectors $\boldsymbol{b}_0, \cdots \boldsymbol{b}_{n-1}$ might not all lie in one plane, as is the case for actual cylinders, the optimization problem above is used to find in a way the most perpendicular vector to all the base vectors. Finding the vector $\boldsymbol{v}$ is a quadratic optimization problem with a quadratic constraint—the unit length of $\boldsymbol{x}$. If $\boldsymbol{v}$ is a solution, then $-\boldsymbol{v}$ is also a solution. We are not aware of any exact solution algorithm that finds the vector $\boldsymbol{v}$ in polynomial time. In our case, however, an approximate solution is sufficient. One way to solve it approximately would be to use *constrained stochastic gradient descent* [Roy and Harandi, 2017].

In our implementation, we simply generate a certain number of vectors lying on a unit half-sphere, compute the optimization function for each of them, and choose as a result the vector with the minimal value of the optimization function. Our implementation evaluates 220 vectors that are generated using the equation

$$f(\theta, \phi) = \begin{pmatrix} \sin\phi\cos\theta \\ \sin\phi\sin\theta \\ r\cos\phi \end{pmatrix}$$

with $\theta \in \{0 \cdot 2\pi/20, 1 \cdot 2\pi/20, \ldots, 19 \cdot 2\pi/20\}$ and $\phi \in \{0 \cdot \pi/20, 1 \cdot \pi/20, \ldots, 10 \cdot \pi/20\}$.

Splitting a potential VGS along an edge $e_i$ means that it is divided into two smaller VGSs; the first consists of triangles $t_0, \ldots, t_i$, the second of triangles $t_{i+1}, \ldots, t_{n-1}$. If a VGS that forms a loop is split along an edge $e_i$, we keep the VGS intact and only mark the VGS as disconnected along the edge $e_i$ and look at it as no longer forming a loop.

The whole VGS detection process begins with identifying potential VGSs, as described earlier. After that, the individual potential VGSs are processed in the following series of steps:

1. If a potential VGS has three or fewer triangles, we discard it.

2. If a potential VGS has some dihedral angles of its interior edges positive and some negative, we split it along one of the edges $e_i$ that are next to another edge with an opposite dihedral angle sign. The two new VGSs are processed subsequently using the same series of steps.

3. For a potential VGS, we compute the sum of the dihedral angles of its interior edges. If we process a potential VGS and the VGS has this sum lower than a given threshold $\eta$, we discard it. Here we are essentially discarding flatter generalized triangle strips.

4. The scale direction $\boldsymbol{v}$ is computed for the VGS as described earlier in this section.

5. We take sliding windows consisting of three successive triangles in the VGS triangle sequence. A VGS forming a loop has two more sliding windows $(t_{n-2}, t_{n-1}, t_0)$ and $(t_{n-1}, t_0, t_1)$, as opposed to a VGS not forming a loop. For a sliding window, we compute its base vector $\boldsymbol{b}'$ as a sum of the base vectors of its three triangles. If the difference between the right angle and the angle between $\boldsymbol{b}'$ and $\boldsymbol{v}$ is greater than an angle threshold $\delta$, the VGS is split along one of the interior edges of the triangle sliding window. The newly split VGSs are processed subsequently with the same series of steps. Here we check if the computed scale direction $\boldsymbol{v}$ is close to a scale direction that might correspond to a given triangle sliding window. A VGS might consist of a large number of triangles and also twist a lot along the way. It might not be possible to appropriately scale down the whole VGS using only one matrix transform. This step detects such twisting and prevents it by recursively splitting the VGS.

6. The scale factor $s$ is computed. A scale factor $s'$ is first computed for each three-triangle sliding window as defined in the previous step. For a sliding window $t_i, t_{i+1}, t_{i+2}$ we compute the following values:

$$
\begin{aligned}
l' &= \sum_{i=0}^{2} \|(\text{midpoint of } e_{i+1}) - (\text{midpoint of } e_i)\|_2, \\
\boldsymbol{h}' &= \frac{1}{3} \sum_{i \in \{1,2\}} (e_i \text{ as a vector}) + \frac{1}{6} \sum_{i \in \{0,3\}} (e_i \text{ as a vector}), \\
\varphi' &= \sum_{i \in \{1,2\}} (\text{dihedral angle of } e_i) + \frac{1}{2} \sum_{i \in \{0,3\}} (\text{dihedral angle of } e_i),
\end{aligned}
$$

where $l', \boldsymbol{h}', \varphi'$ are the estimated arc length, height, and arc angle of a resembled cylinder arc, respectively. When computing $\boldsymbol{h}'$, the vectors from edges should have the same direction. The dihedral angles of VGS boundary edges are considered zero when computing $\varphi'$.

The estimated radius of the cylinder arc is computed as $r' = l'/\varphi'$. Given a parameter *desired radius to height ratio* denoted as $\lambda$, the resembled cylinder arc after the scale-down transformation should satisfy this ratio $\lambda$ or be even flatter to prevent noticeable volume shrinkage. The scale factor $s'$ is computed in the following way:

$$
\begin{aligned}
\boldsymbol{h}'_{proj} &= \boldsymbol{v}(\boldsymbol{v} \cdot \boldsymbol{h}'), \\
\boldsymbol{h}'_{perp} &= \boldsymbol{h}' - \boldsymbol{h}'_{proj}, \\
s' &= \frac{\sqrt{(r'/\lambda)^2 - \|\boldsymbol{h}'_{perp}\|_2^2}}{\|\boldsymbol{h}'_{proj}\|_2}.
\end{aligned}
$$

The expression $r'/\lambda$ denotes the desired length of $\boldsymbol{h}'$ after scaling-down. The square root expression from the above equation denotes the desired length of $\boldsymbol{h}'_{proj}$ after scaling down. We are scaling in the direction of $\boldsymbol{v}$ so the transformation does not affect $\boldsymbol{h}'_{perp}$. The coefficient $s'$ is the ratio between the desired and actual length of $\boldsymbol{h}'_{proj}$.

From the coefficients $s'$ for all triangle sliding windows, we select the minimal value. If this value is greater than or equal to 1, that means the VGS does not actually need to be scaled down, so we discard it. It is also possible that the resulting coefficient $s$ has a very low value, or in rare cases, the expression under the square root is negative for some sliding window; in that case, in our implementation, we set $s$ to be 0.05.

What follows is a short discussion about our approach for construction of the VGSs. In principle, we set out and try to ensure that in the input to the fairing equation there are only VGSs that resemble cylinders that are to some degree flat. The volume shrinkage of the VGSs, resembling flatter cylinders, should be almost unnoticeable.

Our description of the construction contains implementation details such as values for boundary constants, the size of the triangle sliding window, or the

process of generating potential scale direction vectors. We do not consider these implementation details a core part of our algorithm.

The reason why a triangle sliding window is used is that computing values over only one triangle might lead to noisy estimations, and computing values over the whole VGS might result in some areas of the VGS to not be sufficiently scaled down to prevent visible artifacts.

Another way to compute the VGS scale direction $\boldsymbol{v}$ is to take an average of vectors from interior edges $e_0, \ldots, e_{n-1}$. This means that the scale direction $\boldsymbol{v}$ would be in some cases more aligned with the estimated height vector $\boldsymbol{h}'$ of a triangle sliding window. During the computation of $c'$, we however take this misalignment into account by projecting the vector $\boldsymbol{h}'$ into the direction of $\boldsymbol{v}$. What we do not take into account is the case when the triangle base vectors $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}$ are too far from being perpendicular to $\boldsymbol{v}$. During the computation of $c'$, we assume for simplicity that the estimated radius $r'$ is constant when scaling the VGS down in the direction of $\boldsymbol{v}$. The radius would change significantly if the base vectors were also scaled down, which happens when they are not perpendicular to the scale direction. To ensure that the radius is very close to constant, we scale the VGS in a direction approximately perpendicular to the base vectors $\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{n-1}$ and even split a VGS if some of its triangle sliding windows do not have perpendicular enough base vectors. The approach of computing $\boldsymbol{v}$ as the average of vectors from edges $e_0, \ldots, e_{n-1}$ could in some cases lead to non-perpendicular base vectors, as might be the case for a VGS being an open triangle fan that represents a part of a cone with base edges lying in one plane.

## 3.5 Subdivision and weight computation

### Polynomial patches

In our algorithm, polynomial patches are used to estimate the surface over a triangle or two triangles of the original mesh neighboring a seam. Utilizing the patches, we compute the positions of *imaginary* vertices used as conditions in the fairing equation. The imaginary vertices define how the surface continues beyond the seams, which in turn affects how the mesh, as a result of the fairing equation, will look near the seams.

In the article by Nagata [2005], from which we borrow the algorithm for quadratic curve construction, polynomial patches over triangles are quadratic and their coefficients are computed explicitly from the coefficients of the three quadratic curves corresponding to the triangle edges. The patches interpolate the three quadratic curves. In our approach, however, the polynomial patch should also potentially interpolate a cubic spline of a seam edge. Because of that, the patches of our algorithm are cubic. A patch has the following form:

$$Z(x, y) = \sum_{i+j \leq 3} \boldsymbol{a}_{ij}\, x^i y^j,$$

where $i, j \in [0, 3]$ are natural numbers. The polynomial has in total 10 coefficients $\boldsymbol{a}_{i,j}$ which are determined by solving a system of linear equations. All equations involve an interpolation of a certain point.

In the case of interpolating a single triangle $t_i$, we have 3 equations for the interpolation of the three vertices of the triangle. We also take cubic curves

$c_1, c_2, c_3$ parametrized by $t \in [0, 1]$ corresponding to the edges of $t_i$ taken either from a spline segment if an edge is on a seam path or from a quadratic curve respecting vertex normals constructed over an edge otherwise. For each curve $c_i$, we compute two points at $t = 1/3$ and $t = 2/3$. This gives us additional 6 equations for point interpolation. The patch system for a single triangle is therefore underdetermined, having 10 unknowns and 9 equations.

Another case is interpolating two triangles that share a seam edge. This is done for the boundary edges of the VGSs, where we want the surface to be smooth but at the same time not connected topologically as the VGS part needs to be scaled down. Similarly to the previous case, we take the curves $c_1, c_2, c_3, c_4, c_5$ corresponding to the edges of the two joined triangles. For the curve corresponding to the seam edge, we compute the points at $t = 1/3$ and $t = 2/3$; for other curves, we only take one point at $t = 1/2$. This gives us a total of 6 equations for the curve points and 4 for triangle vertices, which results in a square matrix linear system. It is, however, possible for two edges of the neighboring triangles to be in line with each other, giving a total of 5 points to interpolate on the same line of the patch parameter space (we discuss the parameter space later). Therefore, in some cases, the system might not have a solution; it may be impossible for a cubic patch to interpolate all of the points.

To solve the system of equations for cubic patches, we use SVD decomposition. For an underdetermined system, the decomposition gives the least norm solution, and for an overdetermined system, it computes the least squares solution [Golub and Reinsch, 1971]. The system is solved for each coordinate $x, y, z$ of $\mathbb{R}^3$ separately.

What remains to be said is what parameters $x, y$ we set for each point we interpolate. For the main seam vertices around which we later compute imaginary vertices, we use the coordinates $(0, 0)$ an $(1, 0)$. For one of the other triangle vertices $v_i$, we define its projection to the seam edge as the point on the seam edge line with the closest distance to $v_i$—we denote it as $p_i$. For the $x$ coordinate of $v_i$, we take the distance from $p_i$ to the seam edge point with coordinates (0,0) divided by the seam edge length. The $y$ coordinate of $v_i$ is computed as the distance from $v_i$ to $p_i$ divided by the length of the seam edge and optionally multiplied by $-1$ if the vertex is beyond the seam edge (this depends on from which side of the seam edge we compute the patch). The coordinates of points on curves are linearly interpolated between the corresponding curve edge vertex coordinates with a coefficient $t$ that is equal to the coefficient used for the curve point computation.

There are cases where a curve might deviate too far from its corresponding edge. Because the point parameters are linearly interpolated, this might make the patch that interpolates the curve twist in an unpredictable way. In our implementation, we compute the curve deviation for each interpolated point $p$ of an edge $e$ of an interpolated triangle $t$. The deviation is the angle between a vector going from $v$ to $p$ and a vector corresponding to $e$. If this number is greater than any of the two interior angles near $e$ of the interpolated triangle $t$ multiplied by a threshold parameter $\gamma$, we consider the point $p$ deviant. In our implementation, we set $\gamma$ to $1/2$. If there are any deviant points in a patch, linear interpolation of triangle vertices is used to compute the patch—the patch is a plane. For two neighboring triangles, we linearly interpolate both triangles and

then average both sets of coefficients to create a patch represented by a plane for the two triangles. Linearly interpolated patches are also used for the computation of shadow vertices.

A problem that arises with this approach is that if the seam vertices lie on a cubic spline and the imaginary vertices next to them lie on a linear (plane) patch that contains the seam edge, the imaginary vertices are not a smooth extension of the mesh surface. To compensate for this when evaluating an imaginary vertex near a seam edge with a linear patch, we add to the resulting vertex position the difference between the edge cubic curve evaluated at $t = x$ and a linear interpolation between the edge vertices also evaluated at $t = x$, where $x$ is the $x$-coordinate of the imaginary vertex.

### Subdivision

Before the subdivision stage, our algorithm has already performed the detection of sharp edges of the mesh along with the detection and processing of VGSs. In addition, the algorithm has computed the splines of all seam paths.

We first describe how to perform a simple uniform subdivision and then discuss how to modify the subdivision for seams. Uniform subdivision divides all triangles of the mesh into smaller, same-sized triangles that, for an original triangle $t_i$, lie on the same plane as $t_i$ and have the same values of interior angles as $t_i$. An example of uniform subdivision can be seen in figure 3.11. As a subdivision density parameter, we use the number of new smaller edges that are created during the subdivision in place of an original edge. All new vertices that are created by the subdivision have degree 6.

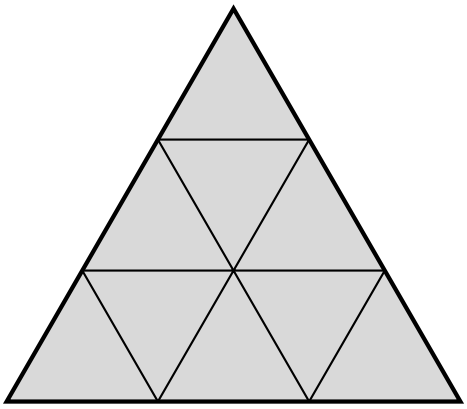


Figure 3.11: A large triangle marked by thicker edges uniformly subdivided into 9 smaller triangles.

The seams are incorporated into the subdivision in the following way: All of the newly created and original seam vertices are duplicated, with the exception of the original vertices that had only one seam edge adjacent to them. One half of the duplicated seam vertices is connected to triangles on one side of the seam, and the other half to triangles on the other sides. During the subdivision, we compute positions for the new seam vertices using cubic splines of the corresponding seam paths. In addition, we save the positions of the seam vertices computed by simple uniform subdivision and mark them as the positions of shadow vertices.

### Polynomial evaluation

During the subdivision, vertices on seams are constructed by evaluating cubic curves that are parts of cubic splines. Given a subdivision density parameter $n$, a straightforward solution is to acquire the seam vertices by evaluating a curve at points $1/n, 2/n, \ldots, (n-1)/n$ if the curve is parametrized with $t \in (0, 1)$. A problem with this approach is that the cubic curve does not have a tangent of constant magnitude, and therefore the final vertex positions might be shifted along the seam, although still on the cubic curve. This creates inconsistent triangles near the seam with potentially very sharp inner angles. A constant magnitude of the curve tangent could be achieved by arc-length reparametrization of the curve.

Arc-length reparametrization of a cubic curve cannot be computed analytically [Wang et al., 2002]. This means that it cannot be expressed as a combination of elementary functions; it can only be evaluated numerically. The numerical evaluation of an arc-length parametrized cubic curve is a valid approach in this case; our solution to the problem is, however, different.

Because cubic curves in our algorithm are running along edges that are line segments, a preliminary point can first be computed on the line segment. This can be done by linear interpolation of the vertices of the corresponding edge. For a seam edge, the preliminary points correspond to the positions of vertices created by uniform subdivision of the seam. A desired point on the cubic curve is then the intersection of the cubic curve with a plane containing the preliminary point on the line segment that is perpendicular to the line segment. This does not correspond directly to points evaluated using arc-length parametrization but gives us points that are, in a sense, very close to vertices that would be created by uniform subdivision. The process leads to more consistent triangles at the boundary and prevents the shifting of vertex positions along the seam.

Instead of computing the exact intersection of the plane and the cubic curve, we compute an approximation of the intersection using binary search. For a curve $P(t)$ parametrized by $t \in (0, 1)$ and a plane $m$, we start with an interval $I \coloneqq (0, 1)$ for the values of $t$. Using the plane normal and a point lying on the plane, we can easily check if a point is above or below the plane. At the start of the algorithm, we assume that $P(0)$ is below the plane and $P(1)$ above the plane $m$. If the curve evaluated at a point in the middle of the interval $I$ (0.5 in the case of $I = (0, 1)$) is below the plane, we set $I$ to the upper half of the interval for the next step of the algorithm ($I \coloneqq (0.5, 1)$ for $I = (0, 1)$). Similarly, if the curve evaluated at a point in the middle of the interval is above the plane, we set $I$ to the lower half of the interval ($I \coloneqq (0, 0.5)$ for $I = (0, 1)$). We repeat this division of the interval $I$ a certain number of times and, in the end, return the middle point of the final interval as the resulting approximation of the intersection of $P(t)$ and $m$.

The same problem arises when interpolating polynomial patches. Because the parameter space is 2D, we cannot use the exact same solution as for cubic curves. In our case, we interpolate imaginary vertices near seams. Four points lying on the seam edge evaluate correctly without shifts along the patch because they correspond to interpolatory constraints used when computing the patch coefficients. These four points with correct evaluation should be enough to prevent noticeable $x$-coordinate shifts on the seam edge and in its close proximity. There is still, how-

ever, a problem with the $y$-coordinate shifts. An evaluated point might be closer or further from a seam edge than what is expected from a given $y$-coordinate. From how we computed the polynomial patch, a $y$-coordinate should approximately correspond to the shortest distance between the evaluated point and the seam edge divided by the length of the seam edge. This can again be solved by using binary search. Similarly to the cubic curve case, we have an interval for potential values of the $y$-coordinate. The $x$-coordinate is fixed during the search. Depending on whether the distance from the evaluated point to the seam edge divided by the seam edge length is greater or less than a desired $y$, we take as the next interval the lower or upper half of the current interval, respectively. As an initial interval, we choose $I \coloneqq (0, 1)$. It is possible that the final value of the $y$-coordinate might be greater than 1. To account for that, the interval is at the beginning of the binary search iteratively enlarged from $(x, y)$ to $(x, 2y)$ until the evaluation of the upper value of the interval results in a greater distance from the seam edge divided by the length of the seam edge than the desired $y$ coordinate.

Another problem to address is the evaluation of the polynomial patch when the evaluated imaginary points are a part of a VGS. After the evaluation, the imaginary points are transformed by a scaling-down matrix $\boldsymbol{T}$ corresponding to a given VGS. What we want from the imaginary vertices is to be about the same distance from a seam as the newly added free vertices on the other side of the seam. That way, triangles around the seam are more similar to each other; in the case of dissimilarity, there is a chance for the resulting surface to create artifacts in the form of surface oscillations. The desired distance from the seam edge is encoded in the parameter $y$ of the polynomial patch. By choosing the right $y$ we are making sure imaginary vertices have the same distance from the seam edge as the free vertices on the other side. After the transformation by $\boldsymbol{T}$, however, these distances might no longer be similar. This is the case if the vector between the imaginary vertex and its projection to the seam has a different direction than the vector between the free vertex and its projection to the seam. The vectors are scaled-down to a higher degree if they are more in line with the scale direction $\boldsymbol{v}$ of a given VGS. To keep the distances of the imaginary vertices from the seam after the scaling-down similar to the free vertex distances, we do the following: Based on the positions of free vertices, we compute the desired distance of an imaginary vertex from a seam edge *after the transformation by* $\boldsymbol{T}$. This distance divided by the length of the seam is used as a parameter $y'$ for a slightly modified evaluation of the imaginary vertex. In this case, we use the evaluated position in the middle of the interval $I$ *with applied transformation by* $\boldsymbol{T}$, when comparing it with $y'$ to determine whether to take the lower or upper half of the interval $I$ in the next step. This should ensure that the resulting imaginary vertex after the transformation by $\boldsymbol{T}$ has approximately the same distance from the seam as transformed free vertices.

### Imaginary vertices

In the section 3.2, we described an algorithm for computing the coefficients of the fairing linear system by following a path at most three vertices long across the subdivided mesh from a newly added free vertex. If a seam vertex $v_i$ is encountered as the second vertex of the path, its mixed Voronoi area needs to be computed, along with the positions and weights $w_{ij}$ of its neighbors for all

possible continuations of the path. The vertex $v_i$ is topologically located at a boundary. Some of the possible continuations of the path from $v_i$ lead back to the existing triangle mesh vertices. Imaginary vertices need to be computed on the spot if the path leads beyond the seam.

From equation 3.1 it can be seen that if a path ends in a non-free vertex, all of its accumulated weights are added to the coefficient $b$ and therefore do not affect the matrix of the linear system. We can choose arbitrary values as positions of imaginary vertices when a path does not end in a free vertex, and it should not affect the symmetry and positive (semi)definiteness of the system.

There is another case where a path starts at a free vertex, continues through a seam vertex, and then ends at a free vertex. Among the accumulated weights of the path is the Voronoi area of the seam vertex, which might depend on positions of imaginary vertices, and also the weights corresponding to both edges of the path, which, in some cases, might also depend on positions of imaginary vertices. Because of this, we cannot be certain that the matrix of the resulting system is positive definite—imaginary vertices do not create a consistent mesh topology, which is a precondition for the proof of positive definiteness we reference in the description of our algorithm (see section 3.2). As we will see later in this section, if the seam vertex in the middle of the path is not an original vertex, the imaginary vertices are constructed in such a way that they create a consistent topology behind the seam. This should not break positive definiteness, as in this case we are essentially putting fixed points for interpolation beyond the seams. An opposite case is that the seam vertex is an original vertex. This case could be eliminated if we fixed all vertices around the original seam vertices using polynomial patch interpolation. The positive definiteness would be guaranteed for the entire algorithm, but at the cost of having to manually set positions for slightly more vertices.

The computation of imaginary vertices described in the following text should not affect the symmetry or positive semi-definiteness of the system.

Another way to look at this part of the algorithm might be not as imaginary vertices on the ends of paths but as a complete neighborhood of a seam vertex $v_i$, with some vertices real and some imaginary. On this neighborhood is computed the mixed Voronoi area of $v_i$ and also, for each imaginary vertex $v_j$, the weights $w_{ij}$, the position of $v_j$, and the position of a shadow vertex corresponding to $v_j$. The mixed Voronoi area of $v_i$ and the weights $w_{ij}$ are computed using the positions of shadow vertices.

We describe the computation of imaginary vertices by considering different cases for a seam vertex $v_i$—note that one seam vertex can have different neighborhoods depending on from which edge $e_i$ it was reached. In the following case descriptions, we mean by original vertices the vertices that were part of the mesh before subdivision or the seam duplicates of those vertices. It is also important to add that we compute the resulting imaginary vertex positions twice (even if this is not always mentioned explicitly in the cases). Standard vertex positions are computed using polynomial patch interpolation, and shadow vertices for cotangent Laplacian weights are acquired using linear patch interpolation. The cases are the following:

1. **Sharp fan** The smooth fan of $v_i$ corresponding to the edge $e_i$ is a sharp fan. The smooth fan does not have a consistent normal, and therefore the

computed normal $\boldsymbol{n}_i$ according to our algorithm is the average of the two normals of the two triangles adjacent to $e_i$. Using this normal, we create 5 imaginary vertices on a plane defined by the normal with the same distance from $v_i$ as is the length of $e_i$. The edges between the imaginary vertices and $v_i$ with the addition of $e_i$ are all 60° apart from each other. By this definition of the neighborhood of $v_i$, we are in a way making sure that the fairing algorithm sees this area as having a normal that is the same as $\boldsymbol{n}_i$.

2. **Not duplicated original seam vertex** The seam vertex is an original vertex that had only one seam edge associated with it. These are the original seam vertices that were not duplicated during the subdivision. The vertex $v_i$ is located at the end of a seam path, where we want the seam path to smoothly transition to the following surface. In this case, we do not create any imaginary vertices but instead use the neighboring vertices of $v_i$ as the final vertices of the path. Only in the case of the two neighboring seam vertices created by duplication sharing the same position, we pick only one of them.

3. **Non-original seam vertex** The seam vertex is not an original vertex. This is the case for a majority of seam vertices. Because uniform subdivision was used, the seam vertex has two neighboring seam vertices and two neighboring vertices created by subdivision inside the original triangles. We keep these four existing vertices and add two imaginary vertices to them. In this case, we try to simulate a consistent mesh topology between the neighboring non-original seam vertices. We depict how this consistent topology looks in figure 3.12. Two neighboring non-original seam vertices both have one imaginary vertex at the same absolute position, forming a triangle $t_j$ above an edge we denote as $e_j$ between them. To make our algorithm simpler, we make the triangle $t_j$ an isosceles with edges next to the imaginary vertex having the same length. As we have stated earlier, our intention is that the distance between the imaginary vertices and the seam is approximately the same as the distance of the free vertices on the other side of the seam form the seam. We define edges $e_k$ and $e_l$ as the edges leading from the two vertices of edge $e_j$ to the same inner vertex. The edges $e_j, e_k, e_l$ form a triangle that shares the edge $e_j$ with $t_j$ but is on the other side of the seam. We estimate the length of the two same length edges of $t_j$ leading to the imaginary vertex as the average of the lengths of $e_k$ and $e_l$. By now knowing, together with the length of $e_j$, the (estimated) lengths of all sides of $t_j$, we can compute the estimated distance of the imaginary vertex of the isosceles $t_j$ from $e_j$ using the Pythagorean theorem and denote it as $h$. We denote as $e'_j$ the original seam edge from which the edge $e_j$ was created by subdivision. The value $h$ divided by the length of $e'_j$ is used as the $y$-coordinate for the evaluation of the polynomial patch corresponding to $e'_j$. For the $x$-coordinate, a corresponding value is used depending on where the middle of the edge $e_j$ is located in the context of $e'_j$. The binary search algorithm described earlier is used for evaluating the final position of the imaginary vertex. The resulting position should have approximately the distance $h$ from the seam. Note that during the computation of the coordinates $(x, y)$ for patch evaluation, only shadow vertices are used. If a

shadow imaginary vertex is computed, a patch created by linear interpolation is employed for evaluation without the addition of the spline difference from the seam edge. If a standard imaginary vertex is computed, a cubic patch is used or a linear patch with an added spline difference in the case of badly conditioned triangles near the seam (what type of patch to use in this case is described previously in this section).

The polynomial patch might be constructed from one triangle in the case of a sharp edge and two triangles in the case of a VGS boundary. In the case of a VGS boundary, the polynomial patch expects the value of the $y$-coordinate to correspond to the distance from the seam after scaling-down. To that end, we multiply the originally computed $y$ coordinate created by dividing $h$ by the length of $e'_j$ by a factor $c$, which is computed as

$$c = \frac{\|\boldsymbol{T}\boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2},$$

where $\boldsymbol{T}$ is the scaling-down matrix of the corresponding VGS and $\boldsymbol{b}$ is the average of the two vectors of edges $e_k$ and $e_l$.

As a summary of this case, we add two imaginary vertices to the neighborhood of $v_i$. An imaginary vertex is created to have approximately the same distance from the seam as the free vertex opposite it on the other side of the seam. Or, more precisely, the edges leading to the imaginary vertex from the seam are created to have approximately the same length as the edges leading to the free vertex on the opposite side. This is done to avoid surface oscillations that might occur when the distances to the seam do not correspond.

4. **Duplicated original seam vertex** In this case, the edge $e_i$ corresponds to an open smooth fan of $v_i$. To construct the neighborhood around $v_i$, we aim to extend the smooth fan to cover the whole area around $v_i$. In equation 3.4, we computed a factor $k$ that represented what fraction of a vertex neighborhood an open fan covers. To make the constructed imaginary vertices have edges with $v_i$ appropriately 60 degrees apart, we compute the number of imaginary vertices we construct to be the maximum of a rounded value of $\left(2\pi k/\frac{\pi}{3} - 1\right)$ and zero. This case will be covered less formally than the previous one. Imaginary vertices are placed uniformly in terms of edge angles into the space where the smooth fan is missing, and their distances from $v_i$ are linearly interpolated from the lengths of the smooth fan boundary edges. These are, however, only preliminary positions that are converted into patch coordinates. We use patch evaluation as in the previous case but linearly interpolate between the values from two patches corresponding to the boundary edges of the smooth fan of $e_i$. The appropriate linear combination of the values from the two patches is the final position of the imaginary vertex. The coordinates $(x, y)$ for the patches are computed from shadow vertices, while adequate linear or polynomial patches are used for vertex evaluation. When interpolating the lengths of the smooth fan boundary edges that are boundaries of a VGS, we compute the scale factor $c$ defined in the previous step for vectors corresponding to the two boundary edges of the smooth fan of $e_i$. The coordinate $y$ is

multiplied by the value interpolated from factors $c$ of both boundary edges before patch evaluation. Note that the interpolation in this entire step is based on how close, angle-wise, an imaginary vertex is estimated to be to each of the boundary edges.

## 3.6 Notes on implementation

During the development of this thesis, parts of the mesh fairing algorithm were changed many times. Probably the most significant change is the move from adding triangles to the mesh that represent the continuation of the surface of the mesh beyond seams to computing the positions of imaginary vertices. However, in our code, there are a lot of remnants of this old approach, so we briefly address it here.

What is computed explicitly in the code and topologically connected to the rest of the mesh are the imaginary vertices of non-original seam vertices, which can be seen in figure 3.12. In addition, there are other out-of-surface vertices that are connected to original seam vertices, these are, however, not used during the computation of linear system coefficients—imaginary vertices of different types replaced them. In the code, all out-of-surface vertices along with their corresponding out-of-surface triangles are called *surface extensions*. Surface extensions are topologically connected to the rest of the mesh during most of the steps of our algorithm. After the surface is smoothed out by solving the fairing equation, the extensions are discarded.

Surface extensions are no longer needed to implement our algorithm, but they provide a way to visualize neighborhoods around certain seam vertices.

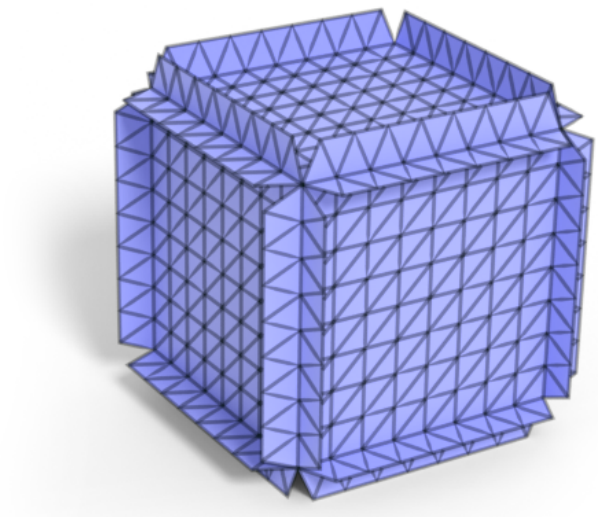


Figure 3.12: Subdivided cube with highlighted edges along with imaginary vertices and their corresponding triangles for non-original seam vertices. The cube had originally two triangles on each side.

# 4. Results

All models referenced throughout this chapter are available in the digital attachment to this thesis under the path `Application/Resources`.

## 4.1 Algorithm options

In this section, we describe the parameters of our algorithm and the corresponding default values. These parameters can be set in the user interface of a desktop application that is part of this thesis. All evaluation in this chapter is performed with the default parameter values unless stated otherwise for a particular case. The default values of parameters in the following list are in parentheses after the parameter name:

**Subdivision type (Laplacian 2)** This option has the following values:

- **Laplacian 1** Minimization using linearized membrane energy. Does not work with shrink prevention.
- **Laplacian 2** Minimization using linearized thin-plate energy. This is the main method of our algorithm.
- **Laplace 3** Minimization using minimal variation. Does not work with feature-retention and shrink prevention.
- **Modified butterfly** An implemented algorithm from an article by Zorin et al. [1996]. The algorithm is not connected to feature-retention.
- **No smoothing** Simple uniform subdivision.

**Edge subdivision count (8)** To how many new edges is each edge of the mesh subdivided. This sets the number of vertices added during the subdivision.

**Number of subdivisions (3)** Similar property to emphEdge subdivision count but it applies only to butterfly subdivision. Two to the power of the option states to how many new edges is each edge of the mesh subdivided.

**Keep sharp edges (turned on)** Whether to detect and preserve sharp edges.

**Edge dihedral angle lower (40 degrees)** The lower angle from section 3.3 for detection of sharp edges.

**Edge dihedral angle upper (60 degrees)** The upper angle from section 3.3 for detection of sharp edges.

**Seam path connection angle (35 degrees)** The maximum angle between directions of two successive edges of a seam path.

**Keep sharp fans (turned off)** Whether to detect and preserve sharp fans. If turned off, the sharp fan case for the creation of seam vertex neighborhood is skipped. Note that because this feature did not show successful results in practice, it is turned off by default.

**Sharp fan average total angle** The threshold angle $\varphi$ from section 3.3 for the detection of sharp fans.

**Shrink prevention (turned on)** Whether to prevent volume shrinkage.

**Max VGS triangle interior angle (20 degrees)** Upper bound for an angle between two inner edges of a VGS.

**Min VGS total dihedral angles (50 degrees)** The threshold $\eta$ for dihedral angles of a VGS (see section 3.4).

**Max VGS base vector deviation (20 degrees)** The threshold angle $\delta$ used for checking that a 3-triangle VGS sliding window has an average base vector approximately perpendicular to the scaling direction $\boldsymbol{v}$ (see section 3.4).

**Desired radius to height ratio (2.0)** The desired ratio between a radius and height of a cylinder resembled by a scaled-down VGS (see section 3.4).

**Keep surface extensions (turned off)** Whether to show surface extensions described in section 3.6 after subdivision. After processing a mesh with this option turned on, no more actions are available on that mesh, as the mesh might no longer be a manifold.

**Keep VGSs scaled down (turned off)** Available only when *Keep surface extensions* is turned on. Does not scale the VGS triangles back to their original sizes after solving the fairing equation. Can be used to look at how VGSs are scaled down.

## 4.2 Speed comparisons

We evaluate the speed of the algorithm on a Stanford bunny model of different sizes. The evaluation was run on a computer with the following configuration:

- **CPU:** Intel Core i7-10870H, 2.20 GHz, 8 cores,
- **RAM:** 16 GB,
- **Memory drive:** SSD,
- **Operating system:** Windows 11, 64-bit.

We do not list the system GPU as it is not used by the algorithm.

Comparisons of algorithm speeds for different types of models are in table 4.1. Note that the actual number of vertices in the linear equation is much higher—table 4.1 shows only the number of vertices before subdividing each edge into 8 smaller edges. For example, the total number of vertices in a subdivided `bunny_large.stl` model is 2 215 298, which is about 64 times higher than the original vertex count.

From table 4.1, it can be seen that for larger meshes, the solution of the linear equation takes most of the time of the algorithm. Therefore, to improve the speed of the algorithm in a meaningful way, the only two options are to either modify the input to the fairing equation or find a faster way to solve the linear system.

| Approach | Number of vertices | | | |
|---|---|---|---|---|
| | 34616 | 10386 | 3463 | 694 |
| Laplace 2 algorithm | 678.8 | 71.7 | 13.5 | 0.96 |
| Laplace 2 solver | 652.1 | 79.7 | 10.9 | 0.27 |
| Laplace 3 algorithm | 2704.0 | 267.8 | 46.8 | 3.58 |
| Laplace 3 solver | 2509.7 | 287.0 | 41.9 | 2.71 |
| Modified butterfly algorithm | 3.78 | 0.89 | 0.29 | 0.06 |

Table 4.1: Times of the execution of our algorithm with either Laplace 2 or Laplace 3 subdivision type or of using modified butterfly subdivision; all other parameters are set to default (although Laplace 3 does not perform feature detection and volume shrinkage prevention). The times are in seconds. The rows corresponding to *algorithm* state the time of the entire algorithm, while the rows corresponding to *solver* state only the time of converting a list of elements with indices to a representation of a sparse matrix for the linear system, factoring the matrix, and substituting the solution for each coordinate $x, y, z$ of $\mathbb{R}^3$. The models used for the evaluation are from the model with the most vertices to the model with the fewest vertices `bunny_large.stl`, `bunny_medium.stl`, `bunny_small.stl` and `bunny_very_small.stl`.

The row in table 4.1 corresponding to *Laplace 2 solver* can be fitted with a quadratic polynomial using polynomial regression, resulting in a polynomial of the form

$$f(x) = -3.59 + 3.23 \times 10^{-3}\, x + 4.54 \times 10^{-7}\, x^2$$

with a corelation coefficient 0.9999756. When fitting the row corresponding to *Laplace 3 solver* with a quadratic polynomial, the polynomial has the following form:

$$f(x) = -7.32 + 9.10 \times 10^{-3}\, x + 1.84 \times 10^{-6}\, x^2$$

with a corelation coefficient 0.99999397. The corelation coefficients are very high in both cases, but there are only four values to fit. A more extensive measurement should take place to approximate the time complexity of the algorithm from the measured data. We show the results of the regressions here to give an idea of how steep the curves for both linear systems are.

When applying linear regression to the row of table 4.1 corresponding to butterfly subdivision, the following form of linear function is obtained:

$$f(x) = -1.12 + 1.11 \times 10^{-4}\, x$$

with a corelation coefficient of 0.9961. This is to be expected as the algorithm only averages vertices in a neighborhood of constant size when computing a newly added vertex.

## 4.3 Visual evaluation

For the evaluation of meshes, we use flat shading as opposed to ray-traced shading with shadows used for other figures in this thesis. That way, potential artifacts might be easier to spot.

### Organic objects

Examples of how our algorithm performs in the case of natural objects without sharp features are shown in figures 4.1, 4.2 and 4.3. Feature retention is turned off in this case in order to clearly examine how well the algorithm smooths out the shape. We are comparing the results to modified butterfly subdivision.

From the examples, modified butterfly subdivision produces significant oscillations of the resulting surface.

Laplace 2 subdivision produces smooth results but, in some cases, suffers from artifacts. On original vertices that are slightly off from their neighboring surface, Laplace 2 subdivision produces artifacts that resemble sharp vertices. This can be seen in detail in figure 4.3. The case of natural objects with a lower number of vertices seems to be a problem for Laplace 2 subdivision.

Laplace 3 subdivision results in very smoothly looking shapes. To our knowledge, there is no mention of Laplace 3 fairing applied to interpolatory subdivision surfaces in the literature. It seems to be well suited for the case of hard to interpolate organic surfaces, even if it is the slowest of the discussed methods. It is important to note, however, that Laplace 3 subdivision suffers from volume shrinkage. This can be seen by running Laplace 3 on the `round_box.stl` model using the application that is a part of this thesis.

### Basic and technically looking shapes

In this part, we demonstrate on examples what our algorithm can do correctly. Some of the mentioned examples contain artifacts, which are addressed later in this section.

Figure 4.4 shows successful results of our algorithm on basic shapes, including meshes prone to volume shrinkage.

In figure 4.5 of a fan disk, detected sharp edges are smoothly interpolated by cubic splines. The sharp edges (up to one seam path not sharp enough to pass hysteresis thresholding) are successfully preserved, and the surface is smoothed only in areas in between the sharp edges.

Figure 4.6 demonstrates that a seam path consisting of sharp edges can smoothly transition into the neighboring surface if it ends in a vertex where there are no other sharp edges. The surface of the object also smoothly follows the seam path curves.

A box in figure 4.7 gives an example of successful prevention of volume shrinkage on a smooth mesh. The volume shrinkage for the same model is shown in figure 3.6

### Problematic cases

**Sharp vertices** Figure 4.8 demonstrates that the issue of volume shrinkage also applies to cones or, in general, to sharp vertices. In this case, our algorithm identifies the closed fan around the tip of the cone as a VGS. The scale factor of the VGS $s$ is set to the lowest possible value. This makes the cone almost flat before being processed by the fairing equation. Let us consider two cases of further development:

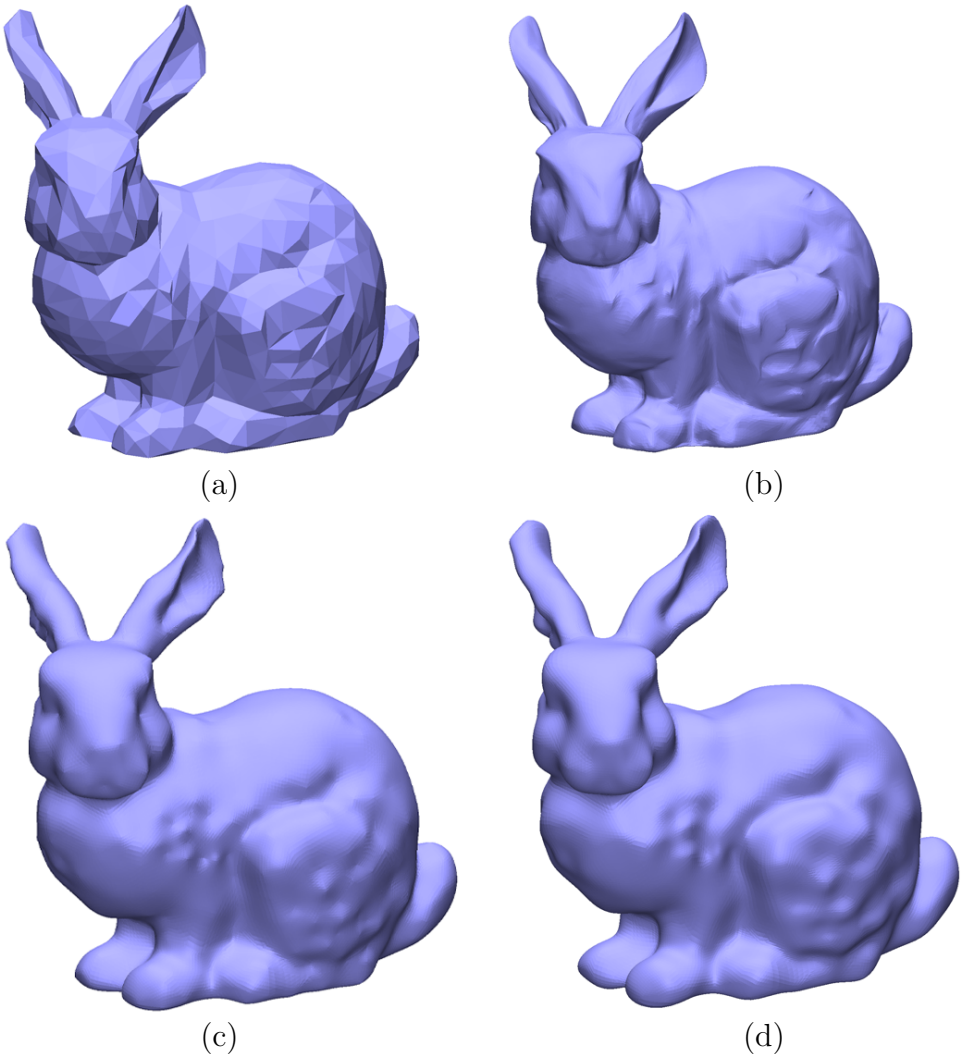



Figure 4.1: Image (a) shows the original model for `bunny_very_small.stl`, image (b) is the result of modified butterfly subdivision, (c) uses Laplace 2 with feature preservation and volume shrinkage prevention turned off, (d) uses Laplace 3.

1. The preservation of sharp fans is turned off. As can be seen on image 4.9 (b), even an almost flat cylinder does not have a fully smoothed out tip vertex after being processed by the fairing equation. The smoothing of a scaled-down regular cone should produce a similar result. After scaling up, the tip becomes more sharp and noticeable, as can be seen in picture 4.9 (c). The expected result is, however, different—that of the tip of the cone being smoothed out. This might be looked at as a form of undesirable volume shrinkage.
2. The preservation of sharp fans is turned on. The fan around the tip of the cone is correctly identified as a sharp fan, and imaginary vertices corresponding to sharp fan neighborhoods indirectly define how the normals close to the tip of the fan should look (see section 3.5). This, however, results in the worst result among the depicted processed cones

in figure 4.9. A form of volume shrinkage occurs here that returns a shape with a very sharp tip instead of the normally looking cone that is the expected result.

Both cases suffer from volume shrinkage. Slightly paradoxically, the expected flat tip after volume shrinkage looks more like a cone tip, and the expected cone tip after volume shrinkage looks like a much sharper tip. To that end, sharp fan preservation is turned off by default in our algorithm. The result is that sharp fans are still preserved due to the volume shrinkage. However, the sharpness of the resulting fans may vary from the sharpness of the input fans, as is the case for image 4.9 (c).

**Ill-conditioned triangles** Another artifact type occurs due to the input mesh triangles having one or two of their inner angles very sharp. This corresponds to the cotangent Laplacian not working well for non-regular meshes with triangles that are very far from being equilateral. We denote these triangles as ill-conditioned. The case can be seen in figure 4.10. In such cases, the badly set weights of the cotangent Laplacian may move the vertices of a subdivided triangle really far from their original position.

**Issues with smooth VGS boundaries** In figure 4.7, it can be seen how the prevention of volume shrinkage works on a mesh with no sharp edges. A problem that sometimes arises near a non-sharp VGS boundary can be seen clearly on image 4.7 (b). A flat square part of the box is slightly round and does not seem to be $G^1$ continuous with the surrounding smooth edges of the box. As we have said in section 3.5 and as can be seen in figure 3.12, our algorithm creates a line of triangles beyond a seam from imaginary vertices that are fixed during the solving of the fairing equation. In the case of a non-sharp seam, the surface around the seam is estimated by one polynomial patch, and from it, two lines of triangles with imaginary vertices are constructed on both sides of the seam. During fairing, vertices on one side of the seam are made to form a smooth surface with the fixed imaginary vertices on the other side of the seam. The main problem with this approach is that the final positions of newly added vertices on one side of the seam might not correspond to the positions of imaginary vertices that were previously there to estimate the same surface. Because the surface from one side of the seam is made to smoothly connect to imaginary vertices on the other side and not to the final positions of vertices, the resulting surface might not be $G^1$ continuous around the seam.

One solution to the discontinuity is to fix all neighboring vertices of the seam by polynomial patch interpolation before solving the fairing equation. This, however, creates another problem when the patch is a linearly interpolated plane. Then all vertices neighboring the seam from both sides would lie in one plane, which could be a noticeable artifact in the final output mesh. Triangles with sharp inner angles are located on both ends of a non-loop VGS, such triangles are more prone to being linearly interpolated (see section 3.5).

Another issue with non-sharp seams is that they in some cases produce surface oscillations, as can be seen in figure 4.11. This might be due to

imaginary vertices forming ill-conditioned triangles on some occasions after scaling down. Several measures were taken in our algorithm to prevent ill-conditioned triangles from forming beyond seams, such as binary search when evaluating splines and patches, accounting for the scaling down during patch evaluation, shadow vertices for weight computation, and the splitting of VGSs that do not have base edges perpendicular to the VGS scale direction. This does not, however, prevent all surface oscillations, and further work in this area is required.

**False positive feature detection on organic models** Figure 4.11 shows the result of running our algorithm with feature detection turned on for a shape that, for the most part, should remain smooth. Some edges are falsely detected as sharp. This might also be the case for sharp fans, whose preservation is, however, turned off by default. A more robust feature detection is required in these cases. It can also be left for the user of the algorithm to manually mark or modify which features are sharp.

## 4.4 Comparisons with other approaches

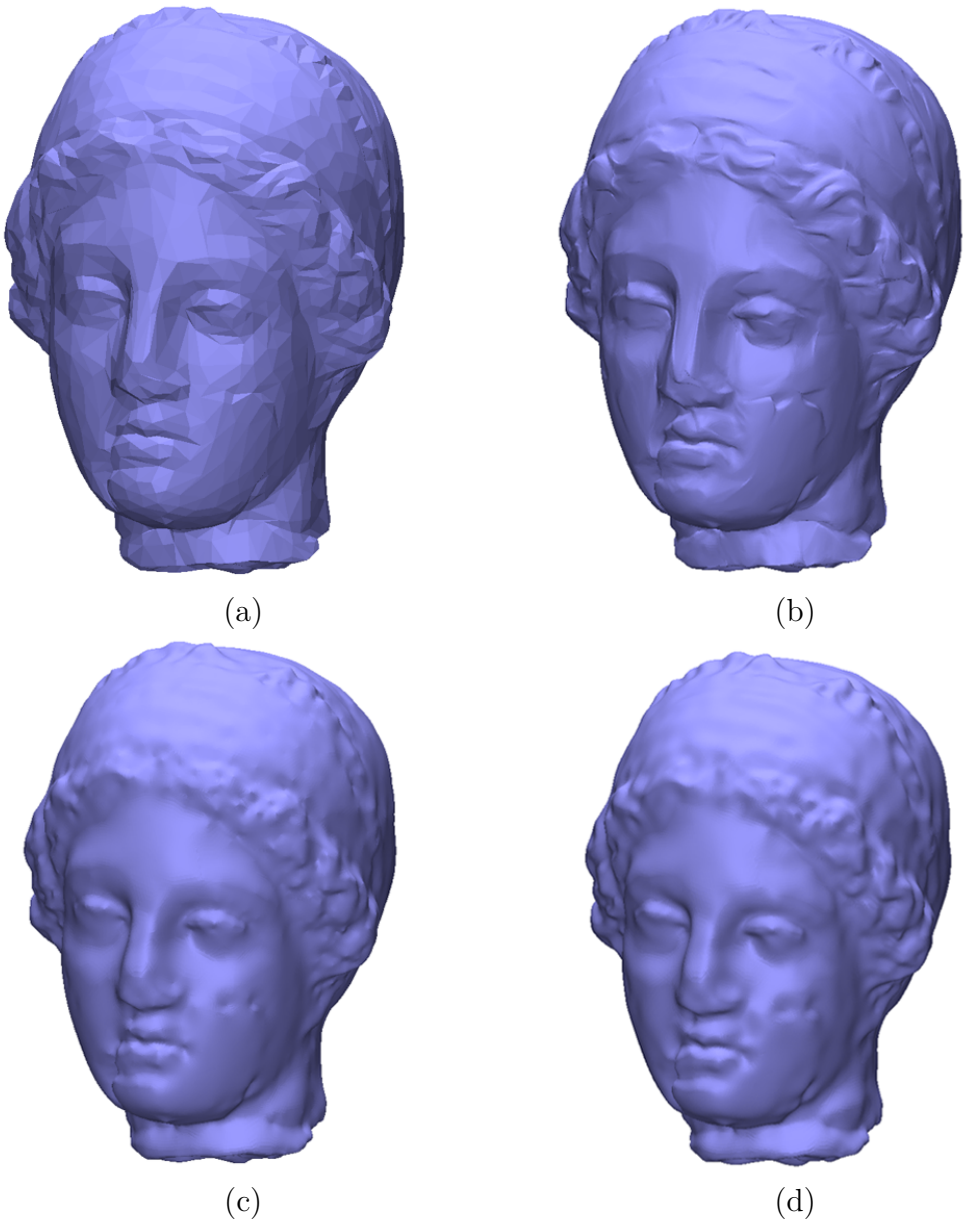
In this section, we compare our algorithm to approaches from chapter 2 that can be applied to interpolatory subdivision. Butterfly subdivision was already covered earlier in this chapter. For the remaining approaches, we only discuss the algorithm differences in text without an actual evaluation on existing models.

### Neural subdivision

The neural network method from an article by Liu et al. [2020] scales linearly with the number of added vertices. In the article, the neural network model is trained on only one triangle mesh. Therefore, it is not clear how well the model would perform when trained over a large mesh dataset for the task of general feature-preserving subdivision. It could, however, introduce fine details into the subdivided mesh, where there is a high probability that the original mesh, from which the coarser input to the algorithm was potentially created, had these details. Our algorithm is incapable of producing such details; it can only be modified to manually add the details, which can be complicated to implement successfully.

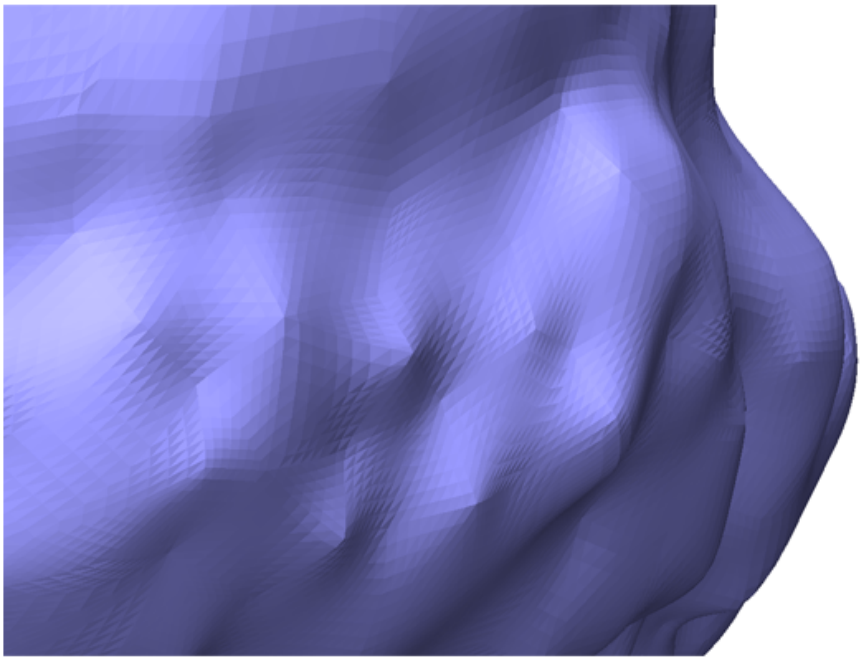
### Polynomial patches

The Bézier patch approach from the article by Su and Senthil Kumar [2005] computes normals in vertices in a similar way to our approach, having cases for sharp vertices and vertices next to sharp edges. It also produces curves with continuous tangents along sharp edges. The approach is linear with respect to the number of input vertices. In theory, quartic Bézier patches should lead to less fair results than a minimization of global fairness energy. We have seen, however, that the minimization of thin-plate energy in some cases produces slightly sharp vertices in places where they should be smooth, which disappear only when minimizing the minimum variation energy.

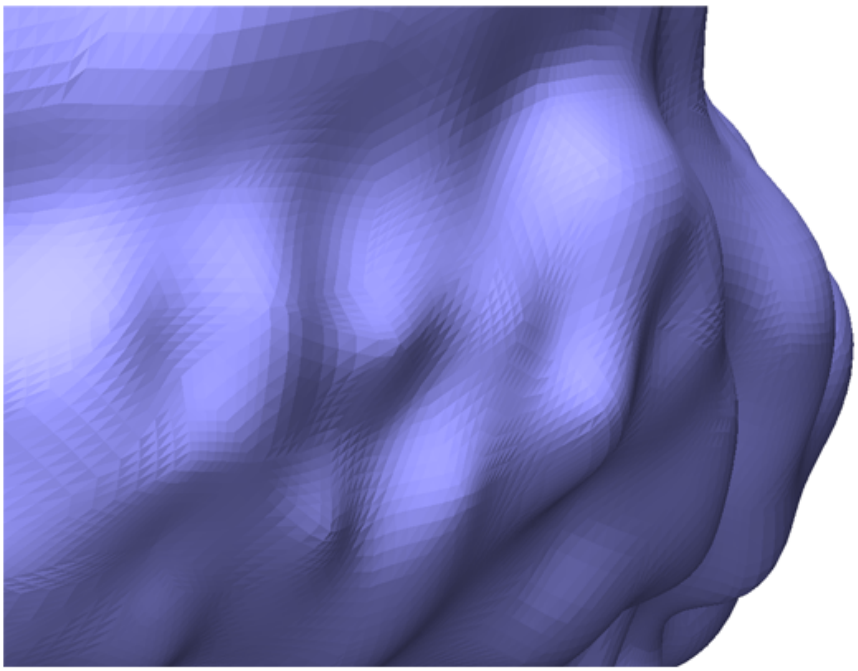


(a) (b) (c) (d)

Figure 4.2: Image (a) shows the original model for `igea.stl`, image (b) is the result of modified butterfly subdivision, (c) uses Laplace 2 with feature preservation and volume shrinkage prevention turned off, (d) uses Laplace 3.



(a)



(b)

Figure 4.3: Both images represent a small area of hair of the `igea.stl` model. Image (a) uses Laplace 2 with feature preservation and volume shrinkage prevention turned off, (b) uses Laplace 3. Notice that some vertices of (a) look sharp.

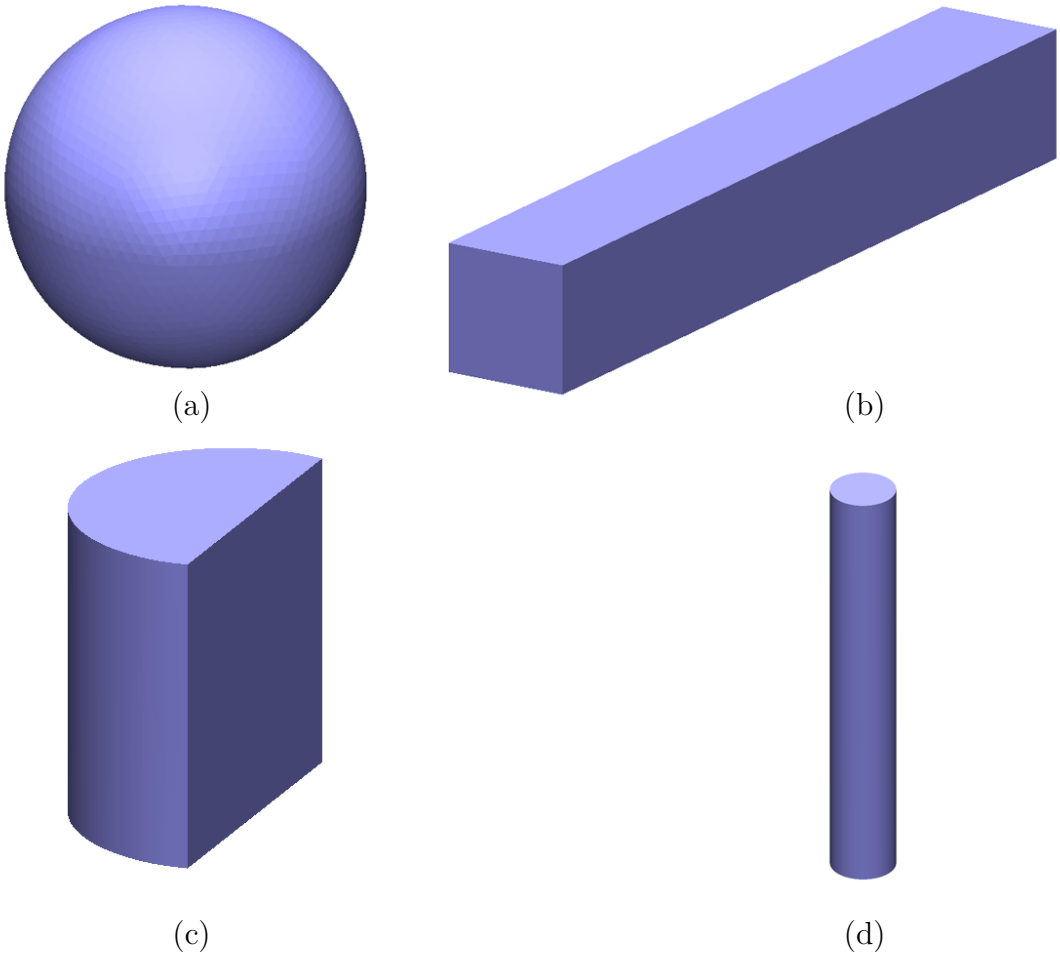
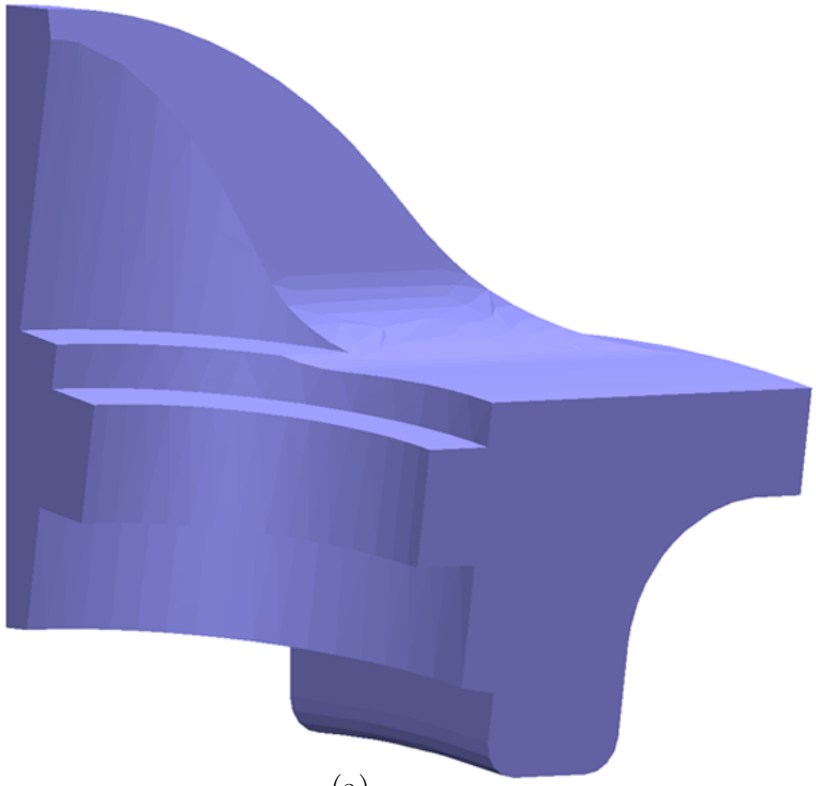
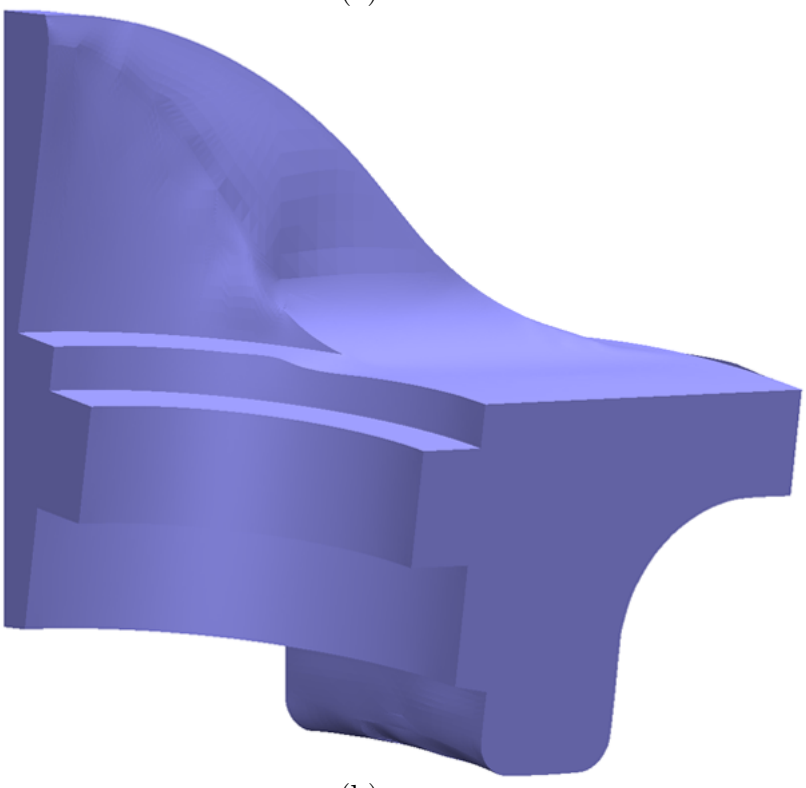



Figure 4.4: The figure shows processed models for `sphere.stl` (a), `rod.stl` (b), `half_cylinder.stl` (c) and `cylinder_higher.stl` (d).



(a)



(b)

Figure 4.5: The image (a) is the original model for `fan_disk.stl`. On image (b) is that model subdivided with default settings of our algorithm. Our algorithm wrongly evaluated a sequence of edges as not sharp. On the bottom of the mesh are slight surface oscillations in places where two VGSs were detected.

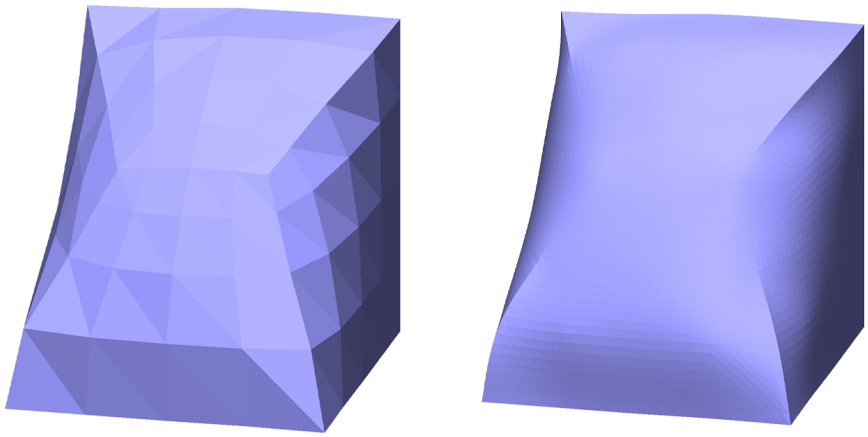


Figure 4.6: On the left is the original model `line_endings.stl`; on the right is that model subdivided with default settings of our algorithm.

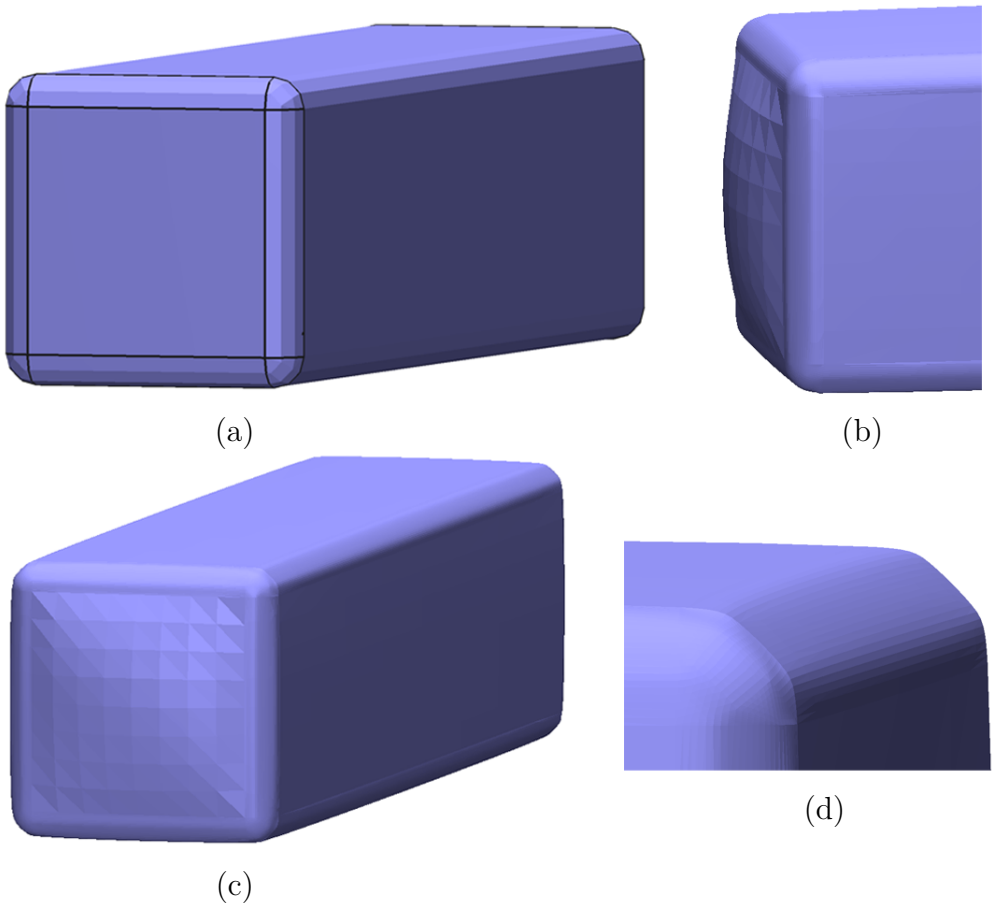



Figure 4.7: On the figure are images of the model `round_box.stl`. The topology of this model can be seen in figure 3.5 (b). On image (a) there are highlighted VGS boundary edges. The mesh has nine VGSs—one for the large middle area of the model and eight for the remaining round edges that are not part of the middle area. Images (a,b,c) show the box subdivided using default settings from different angles. On image (b) it can be seen that the flat square part of the box is slightly inflated. Figure (d) shows the contour of the mesh which is affected by splines computed by the centripetal method.

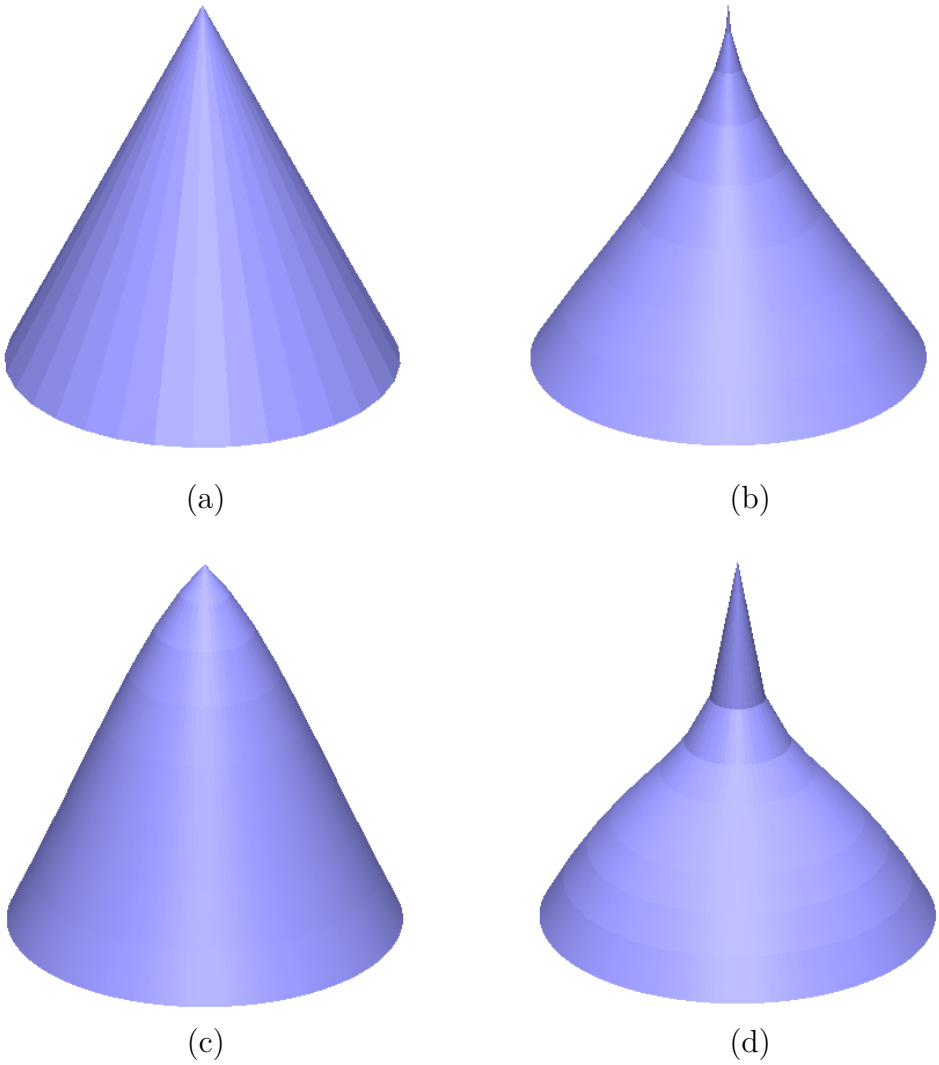



Figure 4.8: Image (a) shows the original model for `cone.stl`, image (b) is the result our algorithm without volume shrinkage prevention, (c) prevents volume shrinkage (the default setting), (d) prevents volume shrinkage and, in addition, retains sharp fans.

Figure 4.9: The figure depicts processed `flat_cone.stl` model from the side. It can be seen that the tip of the flat cone is still visible and was not fully smoothed out.

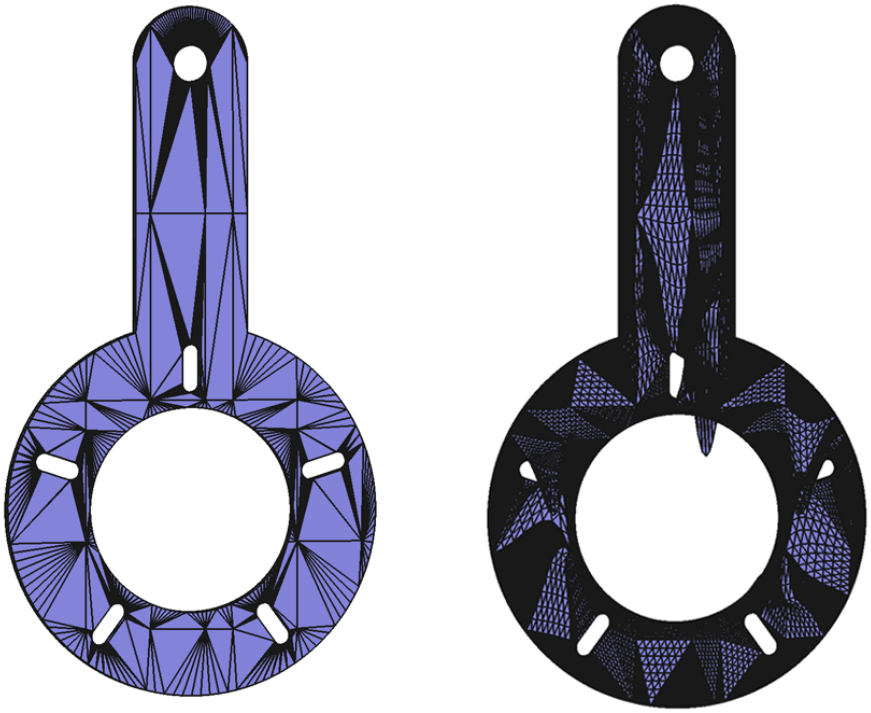


Figure 4.10: On the left is the model `flat_part.stl` with highlighted edges. On the right is that model subdivided with default settings of our algorithm. It can be seen that some of the triangles got moved to areas where previously there was no surface.

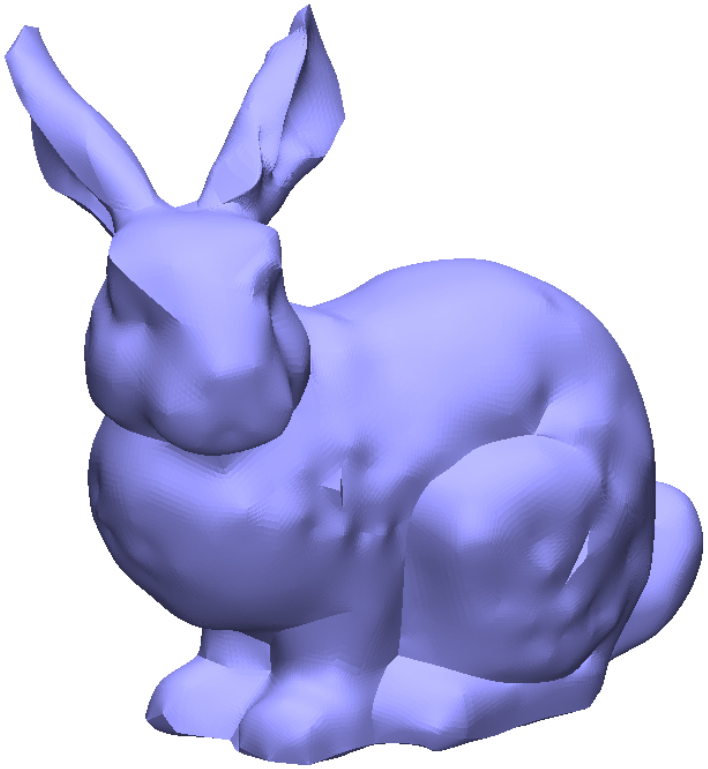


Figure 4.11: The figure depicts processed `bunny_very_small.stl` model. On the right ear of the bunny are surface oscillations caused by a VGS connecting to the remaining surface. Some sharp edges on the surface of the bunny form flat triangles.

# 5. Future work

In this chapter, we cover ways to improve our algorithm either by modifying the existing parts or by replacing some of them with potentially better alternatives.

### Iterative patch-based fairing

When compared to butterfly subdivision [Dyn et al., 1990] or neural subdivision [Liu et al., 2020], our approach takes significantly more time for larger meshes. In this section, we describe a way to decrease the time complexity of our algorithm by iteratively fairing only parts of the mesh with limited size. A triangle mesh before solving the fairing equation could be divided into patches of neighboring triangles with a defined maximum size. All of the patches would be iterated over multiple times. For each iteration, the fairing equation would be solved only for the vertices of the patch. The remaining vertices would be fixed.

This approach might leave some artifacts at the boundaries of the patches, but after multiple iterations over all patches, these artifacts could be eliminated. A way to eliminate these boundary artifacts faster would be to have another set of patches that have different boundaries and alternate between the sets during iterations. The time complexity of the algorithm is linear if we assume the maximum size of the patch and the maximum number of iterations over all patches are constant numbers. The actual execution time of the proposed approach would depend on the values of the previously mentioned constants. There is also an option to process multiple patches that do not neighbor each other in parallel.

The proposed approach might be successful to a lesser degree in cases of mesh hole filling or finding the smoothest deformations. In these cases, there are a large number of free vertices that benefit from being faired as a whole. By dividing the large area of free vertices into patches, it may take a very large number of patch iterations to produce globally smooth results. Because in our case, the global structure of the mesh is already defined by the fixed original vertices, this is not a problem for our task.

The patch-based approach might also solve the surface discontinuity problem of non-sharp seams (see section 4.3). If individual VGSs were also considered patches, both patches on the opposite sides of a seam would, in each iteration, smoothly connect to the vertices of the other patch beyond the seam.

Because the approach is incremental, the results of patch fairing could be viewed by a user in real-time. If the user finds the resulting surface sufficiently smooth, he or she could terminate the algorithm early.

### Adaptive triangle subdivision

To make a surface smooth, a finer discretization is usually needed for parts of the surface with high curvature. On the other hand, flatter parts can be represented with fewer triangles in the resulting smoothed out mesh. To speed up our algorithm while keeping approximately the same level of detail that a uniform subdivision has, each triangle could be non-uniformly subdivided into a particular number of new triangles depending on the curvature around the triangle. The curvature of a triangle might be estimated from the coefficients of a polynomial

patch over the triangle or from angles between directions of normals in the vertices of the triangle. In this approach, it is necessary to define what topology the new triangles would have inside the original triangles and how neighboring subdivided triangles would be connected if one triangle had a higher curvature and therefore more triangles than the other.

### Incremental remeshing

As we have seen in section 4.3, our algorithm suffers from ill-conditioned input triangles. There are techniques to *remesh* the triangle mesh into a more regular mesh with approximately equilateral triangles while retaining the form of the processed shape. A method that could be employed in our algorithm is described in an article by Botsch and Kobbelt [2004]. The method repeatedly splits long edges into two and collapses short edges made of two vertices into a single vertex. In this approach, the edges of the mesh are iteratively becoming more equally sized, which in turn produces more equilateral triangles. It might be possible to adjust the method to keep original vertices and their positions fixed by for example, not collapsing an edge with an original vertex. With this approach applied to our algorithm, the mesh before solving the fairing equation could have well-conditioned triangles, and our method would still remain interpolatory. The method could also split edges to a higher degree on a more curved surface and therefore produce similar results to the adaptive triangle subdivision suggested above.

### Improvement of VGS processing

Some VGSs suffer from surface oscillations after solving the fairing equation. This problem could be analyzed more in depth to design and implement a corresponding solution.

### Improvement of sharp edge detection

Simple hysteresis thresholding is not sufficient for sharp edge detection in certain cases. It could be exchanged for *angle between best fit polynomials* described in section 3.3 or a more robust solution.

### Fixing of certain free vertices

Throughout the thesis, solutions to multiple problems involved the fixation of some free vertices:

1. Fixing of all vertices near non-sharp seams. This is done to prevent the tangent plane discontinuity of the resulting surface around VGSs (see section 4.3).

2. Fixing of all vertices near seam vertices that correspond to original vertices. This is proposed in section 3.5 to make the matrix of the fairing equation positive definite. The positive definiteness of the system might lead to the application of more efficient algorithms to the solution of the linear system.

3. Fixing of all vertices neighboring the tips of sharp fans. This might prevent the sharp fans from becoming undesirably sharper after solving the fairing equation (see section 4.3). The detection of sharp fans is turned off by default in our algorithm—this may no longer be necessary after fixing the vertices appropriately.

For the fixation of free vertices, the polynomial patches proposed in our algorithm might not be sufficiently accurate. One problem is that the patches constructed over a single triangle are not $G^1$ continues between each other over edges. The tangent plane continuity over edges could be guaranteed by using *quartic triangular Bézier patches* proposed in an article by Su and Senthil Kumar [2005].

### Tangent constraints instead of imaginary vertices

The fact that our algorithm needs to define the positions of imaginary vertices beyond seams brings several issues. First, it poses high requirements on polynomial patches that need to evaluate vertices outside of the area over which they were constructed. Second, imaginary vertices lead to tangent plane discontinuities around non-sharp seams, as discussed in section 4.3. Third, the need to define layouts of imaginary vertices for different types of seam vertices adds higher complexity to our algorithm.

According to an article by Jacobson et al. [2010], it is possible to define tangent plane constraints for thin-plate energy minimization without modifying the matrix of the linear system of the minimization. Tangent constraints for the seam vertices might be used in our algorithm instead of the positions of imaginary vertices. We already construct polynomial patches over triangles, so computing tangents to them on the seams would not be difficult. Some details would need to be resolved, however, such as how to deal with sharp fans that do not have a defined normal at their tip.

In section 4.3 we have seen that Laplace 3 leads to very smooth results for organic objects. It, however, requires the computation of edge paths consisting of up to four edges. In our algorithm, this corresponds to having to define paths of length two of imaginary vertices leading from seam vertices. This is the reason why, in our implementation, the preservation of sharp features and prevention of volume shrinkage cannot be used with Laplace 3. From the article by Jacobson et al. [2010] it might, in theory, be possible to define second derivative constraints for boundary vertices in addition to tangent constraints instead of the imaginary vertex paths. Again, the second derivative constraints could be acquired from the already utilized polynomial patches. This would potentially allow for the use of Laplace 3 along with the preservation of sharp edges and volume retention.

# Conclusion

In this thesis, we designed and implemented an algorithm for interpolatory subdivision of triangle meshes with feature-retention. At its core, the method utilizes the minimization of thin-plate energy over the triangle mesh to compute the positions of new vertices added during subdivision. The minimization of this energy is evaluated along with the minimization of the minimum variation energy over selected meshes. Both methods give better results than a traditional subdivision scheme *modified butterfly*, although thin plate energy produces artifacts around certain interpolated points. Minimum variation energy does not have these artifacts but is computationally more expensive. Both methods suffer from the loss of volume of the resulting mesh in some cases.

It is more difficult to build additional constraints around minimum variational energy. Therefore, in the rest of the thesis, we attempted to devise a method that adds feature-retention and volume preservation to an approach that minimizes thin-plate energy. Our proposed method results in a complex algorithm that employs various techniques for feature detection, computation of normals, polynomial spline and patch interpolation, and the topological division of the mesh along some of its edges.

In the evaluation part of the thesis, we showed that our algorithm is capable of volume and feature preservation on selected objects. In addition, we analyzed some of the current drawbacks of the algorithm, such as the impact of ill-conditioned triangles and problems with tangent plane discontinuities along certain edges.

In the future work chapter, these drawbacks are addressed. We suggest a way to modify the mesh to replace the ill-conditioned triangles while still being an interpolatory method. Multiple methods are proposed that try to mitigate the tangent plane discontinuities.

# Bibliography


Michael Bartoň, Iddo Hanniel, Gershon Elber, and Myung-Soo Kim. Precise hausdorff distance computation between polygonal meshes. *Computer Aided Geometric Design*, 27(8):580–591, 2010.

Mario Botsch and Leif Kobbelt. A remeshing approach to multiresolution modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 185–192, 2004.

Mario Botsch and Olga Sorkine. On linear variational surface deformation methods. *IEEE transactions on visualization and computer graphics*, 14(1):213–230, 2007.

Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Lévy. *Polygon mesh processing*. CRC press, 2010.

Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics (TOG)*, 32(5):1–11, 2013.

Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324, 1999.

Nira Dyn, David Levine, and John A Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM transactions on Graphics (TOG)*, 9(2):160–169, 1990.

Michael Garland and Paul S Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, 1997.

Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. *Linear algebra*, 2:134–151, 1971.

Shi-Min Hu, Zheng-Ning Liu, Meng-Hao Guo, Jun-Xiong Cai, Jiahui Huang, Tai-Jiang Mu, and Ralph R Martin. Subdivision-based mesh convolution networks. *ACM Transactions on Graphics (TOG)*, 41(3):1–16, 2022.

Andreas Hubeli, Kuno Meyer, and Markus H Gross. Mesh edge detection. *CS technical report*, 351, 2000.

Alec Jacobson, Elif Tosun, Olga Sorkine, and Denis Zorin. Mixed finite elements for variational surface modeling. In *Computer graphics forum*, volume 29, pages 1565–1574. Wiley Online Library, 2010.

Jürgen Jost and Jeurgen Jost. *Riemannian geometry and geometric analysis*, volume 42005. Springer, 2008.

Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. Abc: A big cad model dataset for geometric deep learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9601–9611, 2019.

Wolfgang Kühnel. *Differential geometry*, volume 77. American Mathematical Soc., 2015.

Eugene TY Lee. Choosing nodes in parametric curve interpolation. *Computer-Aided Design*, 21(6):363–370, 1989.

Chen-Hsuan Lin, Jun Gao, Luming Tang, Towaki Takikawa, Xiaohui Zeng, Xun Huang, Karsten Kreis, Sanja Fidler, Ming-Yu Liu, and Tsung-Yi Lin. Magic3d: High-resolution text-to-3d content creation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 300–309, 2023.

Hsueh-Ti Derek Liu, Vladimir G Kim, Siddhartha Chaudhuri, Noam Aigerman, and Alec Jacobson. Neural subdivision. *arXiv preprint arXiv:2005.01819*, 2020.

Charles Loop. Smooth subdivision surfaces based on triangles. January 1987. URL `https://www.microsoft.com/en-us/research/publication/smooth-subdivision-surfaces-based-on-triangles/`.

Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H Barr. Discrete differential-geometry operators for triangulated 2-manifolds. In *Visualization and mathematics III*, pages 35–57. Springer, 2003.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics (ToG)*, 41(4):1–15, 2022.

Takashi Nagata. Simple local interpolation of surfaces using normal vectors. *Computer Aided Geometric Design*, 22(4):327–347, 2005.

Steven J Owen, David R White, and Timothy J Tautges. Facet-based surfaces for 3d mesh generation. In *IMR*, pages 297–311, 2002.

Ulrich Pinkall and Konrad Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental mathematics*, 2(1):15–36, 1993.

Konrad Polthier. Imaging maths-inside the klein bottle. 2003. [Online; accessed 11-July-2023].

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

Soumava Kumar Roy and Mehrtash Harandi. Constrained stochastic gradient descent: The good practice. In *2017 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 1–8. IEEE, 2017.

Justin Solomon. Course webpage of 6.838: Shape Analysis (Spring 2021). `http://groups.csail.mit.edu/gdpgroup/6838_spring_2021.html`, 2021. [Online; accessed 19-April-2023].

Yi Su and A Senthil Kumar. Generalized surface interpolation for triangle meshes with feature retention. *Computer-Aided Design and Applications*, 2(1-4):193–202, 2005.

Bruno Vallet and Bruno Lévy. Spectral geometry processing with manifold harmonics. In *Computer Graphics Forum*, volume 27, pages 251–260. Wiley Online Library, 2008.

Desmond J Walton and Dereck S Meek. A triangular g1 patch from boundary curves. *Computer-Aided Design*, 28(2):113–123, 1996.

Hongling Wang, Joseph Kearney, and Kendall Atkinson. Arc-length parameterized spline curves for real-time simulation. In *Proc. 5th International Conference on Curves and Surfaces*, volume 387396, 2002.

Zhihao Wang, Jian Chen, and Steven CH Hoi. Deep learning for image super-resolution: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 43(10):3365–3387, 2020.

Max Wardetzky, Saurabh Mathur, Felix Kälberer, and Eitan Grinspun. Discrete laplace operators: no free lunch. In *Symposium on Geometry processing*, pages 33–37. Aire-la-Ville, Switzerland, 2007.

Research, Inc. Wolfram. Mathematica, Version 13.2, 2023. URL `https://www.wolfram.com/mathematica`. Champaign, IL, 2022.

Denis Zorin, Peter Schröder, and Wim Sweldens. Interpolating subdivision for meshes with arbitrary topology. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 189–192, 1996.

# List of Figures

# A. Attachments

## A.1 Application

The implementation of our algorithm is available as a part of a graphical user interface (GUI) application that can be run on the Microsoft Windows operating system. In the application, a triangle mesh can be loaded from a file, subdivided, and observed in 3D. The application is an executable that can be found in the `Application` folder under the name `mesh_processing_app.exe` in the digital attachment to the thesis.

### A.1.1 System requirements

The following are our recommended minimum system requirements to run the application:

- **CPU:** 2 GHz, 2 cores, x64
- **RAM:** 8 GB,
- **Operating system:** Windows 10, 64-bit,
- **GPU:** Supports OpenGL 3.3.

In addition, there should be a Visual C++ Redistributable of version 2019 or 2022 installed on the system. It is available at `https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist`.

### A.1.2 User documentation

Upon start, the application displays a default 3D model located in a 3D space. Different 3D models of the formats `.stl` and `.obj` can be imported into the application or saved after being processed by the application.

#### Menu options

The application contains a menu bar on top of the viewport with the following options:

**File** This submenu allows the import and export of `.stl` and `.obj` meshes. The loaded meshes should only consist of triangles and be manifolds to ensure the proper functionality of our algorithm. There is an incorporated file explorer that, for import, opens in the `Resources` folder next to the application executable. The folder contains a selected collection of meshes, including all meshes shown in figures in this thesis, with the same file names as those used in the text.

**Appearance** In this submenu, a *View mode* determines if the 3D model is rendered using flat shading (*Default* option) or with highlighted edges without any lighting (*Edges* option). The *Sharp edges* checkbox can be checked

to highlight all detected sharp edges (black color). The *VGS boundary edges* checkbox corresponds to the highlighting of VGS boundary edges (red color). If an edge is both a sharp edge and a VGS boundary edge, it is marked black when both checkboxes are checked.

**Subdivision** The *Subdivide* button starts the subdivision with the currently set parameters. Note that it might take some time to subdivide the mesh and during this time the application is not responding to any user input. The options under the *Options* menu correspond to the algorithm options described in section 4.1.

#### Controls

The 3D navigation in this application involves looking at a *center point*. The viewport camera can orbit around the center point, and the point can be moved along with the camera. The navigation has the following controls:

- Dragging a mouse while holding the left mouse button orbits the viewport camera around the center point.
- Dragging a mouse while holding the right mouse button moves the center point along with the camera.
- Mouse wheel or the keys $I$ and $O$ can be used to zoom in and out.
- The key $B$ can be used to reset the center point position and camera angle to their default values.

### A.1.3 Source code

The source code is located in the folder `MeshProcessing` in the digital attachment to this thesis.

#### Setup

The following requirements are needed for a successful project setup:

- CMake of version 3.1 or higher (`https://cmake.org`).
- Git (`https://git-scm.com/`).

The project setup consists of the following steps:

1. Run `setup.bat` script located in the `MeshProcessing` folder, which generates a Visual Studio solution in `build` folder.
2. Open the solution in Visual Studio, build and run selected projects. The path to the solution file is `MeshProcessing/build/mesh_processing.sln`.

The setup script first clones all Git submodules of the project. Among them is vcpkg (https://vcpkg.io) which is used as an internal package manager for some of the third-party libraries connected to the solution. The setup script initializes vcpkg and then calls CMake to create the Visual Studio solution.

### Solution structure

The solution is divided into two main projects:

1. *MeshProcessing*—an internally used library that contains the implementation of our algorithm.
2. *MeshProcessingApp*—an executable application with a GUI that uses the library.

In addition, in the solution is a `Resources` folder that contains GLSL shaders and a collection of 3D meshes.

The solution contains modified copied code of some of the files from PrusaSlicer (`https://github.com/prusa3d/PrusaSlicer`). What we utilize from the PrusaSlicer is an implementation of the half-edge representation of a triangle mesh and methods for importing and exporting models.

Third-party libraries used in the solution are the following:

- eigen (`https://gitlab.com/libeigen/eigen`)
- boost (`https://github.com/boostorg/boost`)
- admesh (`https://github.com/admesh/admesh`)
- cereal (`https://github.com/USCiLab/cereal`)
- oneTBB (`https://github.com/oneapi-src/oneTBB`)
- Catch2 (`https://github.com/catchorg/Catch2`)
- glad (`https://github.com/Dav1dde/glad`)
- glfw (`https://github.com/glfw/glfw`)
- imgui (`https://github.com/ocornut/imgui`)
- imgui-notify (`https://github.com/patrickcjk/imgui-notify`)
- ImGuiFileDialog (`https://github.com/aiekick/ImGuiFileDialog`)
- glm (`https://github.com/g-truc/glm`)

The libraries are either downloaded via vcpkg, cloned as a git submodule, or have copied source code into the solution.