Curvature-Adaptive Remeshing with Feature Preservation of Manifold Triangle Meshes with Boundary

Master's Project

Tanja Munz

Master of Science
Computer Animation and Visual Effects

24th August, 2015

Abstract

In computer graphics meshes are a common representation of surfaces. These meshes are often irregular and contain more vertices than necessary to approximate a surface sufficiently. With remeshing a better mesh can be created for the underlying surface with respect to some quality criteria. In this project, manifold triangle meshes with boundaries and feature edges are considered; they are represented by a half-edge data structure for fast access of neighbouring elements of vertices, edges and triangles. The remeshing approach used is based on local mesh operations (edge-split, edge-flip, edge-collapse and vertex relocation) to adjust the density of elements and to improve their connectivity for a more regular mesh. An optional back-projection of vertices to the original surface can be used to keep the shape of the new mesh close to the reference mesh. With a sizing field based on an approximation of the curvature of the surface it is possible to adjust the sampling density to represent the final mesh with fewer triangles in areas of low curvature. The result of this approach is curvature-adaptive isotropic remeshing of good quality with feature preservation. The whole remeshing process is implemented in an interactive application that allows modifications of parameters.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling

Keywords: Curvature-Adaptive Remeshing, Isotropic Remeshing, Local Mesh Modifications, Triangle Mesh, Feature Preservation, Half-Edge Data Structure

Contents

1. Introduction

1.1. Motivation

Triangle meshes are a common way of representing surfaces in computer graphics. Meshes can be generated in different ways: by modelling, sculpting, deformation or by scanning devices and isosurfacing of implicit representations. Especially meshes that are generated algorithmically are often rarely satisfactory. They are irregular, oversampled and contain redundant vertices.

Application areas for meshes and reasons for remeshing vary and they usually need or create a mesh of a certain quality. Common areas to consider are geometric modelling, shape editing, animation, numerical simulation (finite element simulation), visualisation and storage. All of these areas benefit from optimised representations of meshes.

With remeshing a better mesh for a surface can be determined: the complexity of the mesh can be reduced and the quality can be improved. The first aspect – the reduction of the complexity – is usually handled with mesh simplification. The goal of remeshing is to additionally optimize the mesh in terms of vertex sampling, triangle quality and regularity. Usually, it is desired to have triangles with equal edge lengths. The best representation of a mesh in regard to sampling density is if areas with small details are modelled by more triangles while flat areas are represented by fewer elements. This is often referred to as curvature-adaptive remeshing when the vertex distribution is related to the underlying approximated curvature of the surface. Sharp edges and features are important to be preserved in order to retain the general shape.

Remeshing can be realised by determining a new mesh for the surface or by modifying the original mesh by adding and removing vertices and changing their positions. To modify elements different methods are distinguished: Parameterisation methods create a parameterisation for the mesh in the 2D domain (with global parameterisation for the whole mesh; with local parameterisation for patches) while other methods work directly with the three-dimensional surface.

Parameterisation is usually quite expensive but might create meshes of a better quality. Especially for interactive algorithms it is important that the remeshing algorithm is fast. Therefore the tradeoff between quality of the result and speed of the operations have to be considered. Methods using local modifications in 3D are often preferred for real-time environments.

1.2. Objectives

The objective of this project is the implementation of an interactive application for remeshing of surface meshes. The application has to be able to import meshes and represent them in an appropriate data structure for further processing. The result obtained by the remeshing process should be a new mesh that is either uniform or curvature-adaptive (depending on the settings) and regular. In the user interface specific parameters should be able to be adjusted. Results should be able to be exported for further processing in other applications.

The process for remeshing includes the calculation of an approximated curvature of the surface to allow adaptive remeshing. Features have to be detected to preserve sharp corners and edges in the surface. Using local operations on the surface mesh such as edge-split, edge-collapse and edge-flip a more regular mesh is created. Further, relaxation optimizes the result and a back-projection decreases the error-difference to the original surface. This general process is outlined in Dunyach *et al.* (2013).

1.3. Structure

The remaining work is structured as follows:

Chapter 2 – Related Work: This chapter gives an overview of different techniques used for remeshing.

Chapter 3 – Technical Background: Important terms used in mesh processing are explained and necessary techniques required for the implementation are introduced. This includes general topological and geometrical mesh operations like edge-split, edge-collapse, edge-flip and a relaxation technique but also feature handling and the calculation of the surface curvature.

Chapter 4 – Implementation: Details of the implementation are given in regard to structuring the implementation, the used data structure to represent the mesh and details about the created application.

Chapter 5 – Remeshing Results: Results obtained by the implemented remeshing process are provided and discussed.

Chapter 6 – Conclusion: The work is summarised and an overview of further enhancements of the application and of the remeshing approach in general are given.

2. Related Work

In the following sections there will be an overview of research in the area of remeshing. Many publications exist in this area; only a small selection of them is mentioned below. A more detailed survey and descriptions of some techniques can be found in Alliez *et al.* (2008) and in the book Botsch *et al.* (2010).

Botsch *et al.* (2010) covers all important areas of mesh processing: Among other areas different data structures, remeshing, mesh simplification, mesh repair and mesh deformation are covered.

Many algorithms exist in the area of remeshing. Most of them are limited to specific mesh types. Commonly used polygon meshes are triangle meshes and quad meshes. This project is only dealing with triangle meshes. For quad meshing the survey Bommes *et al.* (2012) should be mentioned. An advantage of triangle meshes is that it is possible to triangulate all polygons while it is more difficult to create quads from triangles.

Another property of a mesh is whether it is manifold or non-manifold. Most algorithms have problems handling non-manifold meshes. Remeshing algorithms exist that explicitly handle non-manifold meshes (e.g. Vivodtzev *et al.* (2005) and Zilske *et al.* (2007)) or are limited to closed manifolds (e.g. Fu and Zhou (2009)). The approach in this project is dealing with manifold meshes (optionally with boundary); this type of mesh can be handled by most remeshing algorithms.

When creating a new mesh for a given reference mesh the new mesh can be created from scratch or by changing the existing mesh. Here, a number of approaches are to be distinguished:
Global parameterisation methods generate a parameterisation of the whole mesh, resample the mesh and then project the new mesh connectivity back into three-dimensional space. This technique is used for instance by Alliez *et al.* (2002), Alliez *et al.* (2003) and Kobbelt and Botsch (2003).
Other methods use no parameterisation, work directly on the mesh in 3D and use local mesh modifications (adding, removing and relocating vertices) to improve the vertex distribution in order to obtain a more regular mesh. Hoppe *et al.* (1993), Kobbelt *et al.* (2000), Vorsatz *et al.* (2001) and Botsch and Kobbelt (2004) are examples that use such methods.
Additionally, there is local parameterisation that generates a parameterisation for small patches of the mesh and performs local modifications. It is used for example by Vorsatz *et al.* (2003), Surazhsky and Gotsman (2003), Surazhsky *et al.* (2003) and Fuhrmann *et al.* (2010).

One of the first remeshing techniques for surfaces is described in Turk (1992). Here, local mesh modifications were used.

The density distribution for meshes obtained by a remeshing approach can be equal for the whole mesh or related to some properties of the mesh. Usually uniform and curvature-adaptive remeshing is to be distinguished. The first one creates an equal distribution of vertices for the whole mesh while adaptive approaches use a density field in order to adjust the sampling of vertices or edge lengths. For the density field usually an approximation of the curvature of the underlying surface is calculated. Publications dealing with uniform remeshing are Vorsatz *et al.* (2003) and Botsch and Kobbelt (2004). Curvature-adaptive remeshing is described for example in Frey (2000), Alliez *et al.* (2002), Surazhsky and Gotsman (2003), Alliez *et al.* (2003), Fuhrmann *et al.* (2010) and Dunyach *et al.* (2013).

In order to handle geometry with sharp edges and corners feature detection and preservation is required (such as in Alliez *et al.* (2003) and Vorsatz *et al.* (2003)). A simple method to detect features is using dihedral angles as by Fuhrmann *et al.* (2010). Other methods exist for extracting features that are less sensitive to noise. Such an approach is mentioned by Lévy *et al.* (2002). Most remeshing methods use feature preservation or it could be added in a simple way. Specific remeshing algorithms exist for dealing with features. For example Vorsatz *et al.* (2001) and Kobbelt and Botsch (2003) use attraction forces to shift vertices into the direction of feature edges. Attene *et al.* (2003) even try to enhance features.

For interactive applications real-time response and hence a fast remeshing algorithm is required. Most of such algorithms only perform uniform remeshing or create results of a lower quality. Not many fast curvature-adaptive algorithms as described by Dunyach *et al.* (2013) exist.

The technique explored and implemented in this project relies on the method described in Dunyach *et al.* (2013). This approach works with triangle meshes and mesh modifications directly on the mesh in three dimensions are used to obtain a more regular mesh of high quality in real-time. It is extended from Botsch and Kobbelt (2004) and Botsch *et al.* (2010), which are creating a uniform isotropic mesh, to an approach for curvature-adaptive remeshing. The curvature of the mesh is calculated in order to adjust the vertex density and features to be preserved are detected. In Dunyach *et al.* (2013) the algorithm is used for an interactive mesh modelling application for mesh deformation and mesh sculpting. Each step of mesh deformation is followed by one iteration of adaptive remeshing.

3. Technical Background

In this chapter the technical background for curvature-adaptive isotropic remeshing will be covered. This includes background information on meshes in general and the goals to be achieved by remeshing. The remeshing process will be described and operations used on surface meshes necessary for the implementation will be explained. Some images and pseudocode will exemplify these techniques and operations.

3.1. Mesh

A mesh can be seen as a piecewise approximation of a smooth surface. Common ways to represent meshes are by using quads or triangles. This project focuses on triangles only.

A mesh is usually defined by a list of vertices, edges and triangles and rules for their connectivity.

A *vertex* represents a position in three-dimensional space and an *edge* is a connection between two vertices. A *face* is a surface, it is defined by an ordered list of vertices (or edges) and a *triangle* is a specific type of face with exactly three vertices (or edges).

Additionally it is possible to add further properties to the mesh. The following we assume that a set of edges and/or vertices are defined as feature elements.

More formally, a triangle mesh with N_V vertices, N_E edges and N_T triangles can be defined as a 3-tuple $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{T})$ of vertices $\mathcal{V}$, edges $\mathcal{E}$ and triangles $\mathcal{T}$.

Vertices: $\mathcal{V} = \{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{N_V-1}\}$ with $\mathbf{v}_i \in \mathbb{R}^3$.

Edges: $\mathcal{E} = \{e_0, e_1, \ldots, e_{N_E-1}\}$ with $e_i = \{\mathbf{v}_j, \mathbf{v}_k\}$ and $j, k \in [0, N_V - 1]$.

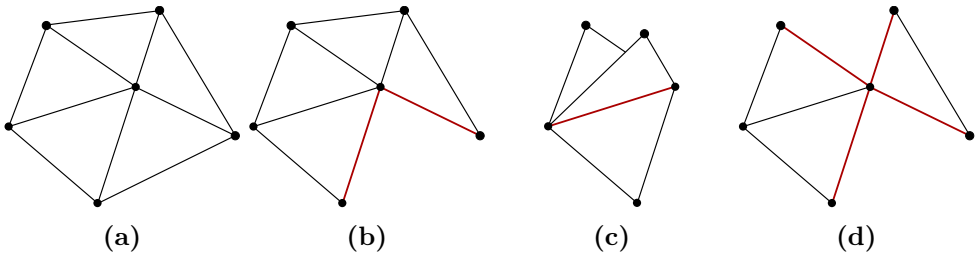


Figure 3.1.: *Different types of a mesh (with red edges of interest): (a) manifold mesh, (b) manifold mesh with boundary, (c) and (d) non-manifold mesh.*

Triangles: $\mathcal{T} = \{t_0, t_1, \ldots, t_{N_T-1}\}$ with $t_i = (\mathbf{v}_j, \mathbf{v}_k, \mathbf{v}_l)$ with $j, k, l \in [0, N_V - 1]$.

Feature edges: $\mathcal{E}_F \subseteq \mathcal{E}$.

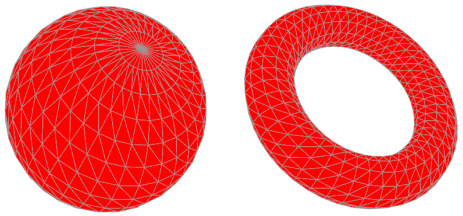
Each *vertex* has a valence, also known as degree of a vertex. It is the number of adjacent vertices (or outgoing edges) to a vertex.
The *one-ring neighbourhood* of a vertex are all vertices that are adjacent to it as well as all incident faces and edges.
Some meshes allow boundaries in their structure. Boundaries are special edges that are incident to only one triangle.

Manifold – Non-Manifold

The use of different data structures and algorithms involves limitations for the representable meshes. Generally it is to be distinguished between **manifold** and **non-manifold** meshes.

A mesh is manifold if each edge is connected to exactly two faces for inner edges (figure 3.1a) or one face for boundary faces (figure 3.1b). For each vertex the incident triangles have to build a closed fan or a fan with one opening if a boundary is allowed.

A mesh is non-manifold if an edge is connected to more than two faces (figure 3.1c) or triangles connected to a vertex do not build a closed fan (figure 3.1d) as previously mentioned. These meshes are suboptimal for most algorithms.



(a) (b)

Figure 3.2.: *Objects with different genus: (a) sphere with genus 0 and (b) torus with genus 1.*

Genus

The genus specifies the number of holes in a closed orientable mesh. A sphere has genus 0 as there are no holes as visible in figure 3.2a, genus 1 meshes have one hole (figure 3.2b), an example is a torus. Meshes with genus larger than 0 cause problems for some mesh algorithms.

Euler Characteristic

The Euler characteristic shows a relation between the number of vertices N_V, the number of edges N_E, the number of faces N_F and the genus g in a closed mesh. It is given as:

$$N_V - N_E + N_F = 2 - 2g$$

According to this formula some properties for meshes can be derived:

- The number of triangles in a mesh is roughly twice the number of vertices: $N_F \approx 2N_V$.
- The number of edges in a mesh is roughly three times the number of vertices: $N_E \approx 3N_V$.
- The average vertex valence is 6.

3.2. Remeshing

In Alliez *et al.* (2008) remeshing is defined in the following way:
"Given a 3D mesh, compute another mesh, whose elements satisfy some quality requirements, while approximating well the input".

Quality can be related to different criteria such as sampling, grading, regularity, size and shape of elements (Alliez *et al.* 2008). Usually the goal is to satisfy a combination of these criteria. Some remeshing approaches create a mesh for the surface from scratch while others change vertex positions and the connectivity in order to achieve a better mesh.

3.3. Mesh Properties

The following aspects have to be considered when choosing a remeshing approach.

Most common *element types* are triangles and quadrangles (figure 3.3a); different algorithms are limited to specific element types.

For the *shape* of elements usually a distinction is made between isotropic and anisotropic elements (figure 3.3b). A triangle for instance has a higher isotropy the closer it is to equilateral. If triangles are close to equilateral edge lengths are almost equal. Isotropic elements are preferred for example for numerical simulations.

The *density* of elements should represent a surface in a satisfying way. Uniform and non-uniform/adaptive element distributions are possible. For uniform remeshing elements are evenly spread across the mesh. For adaptive remeshing the number of elements varies. Supposed, curvature-adaptive sampling is desired areas with low curvature have fewer and larger elements while areas with high curvature are represented by more and smaller elements. Usually fewer elements are required for adaptive remeshing than for uniform remeshing to achieve a comparable approximation. Using uniform sampling many elements would be needed to represent small details.

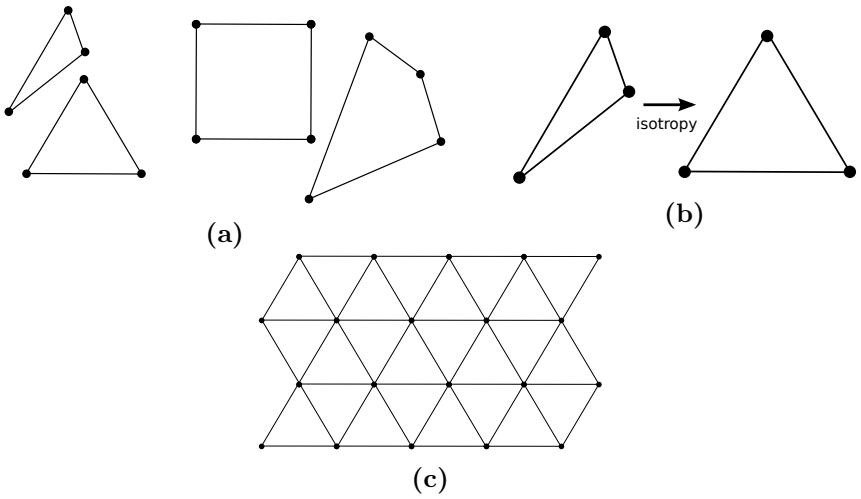


Figure 3.3.: *Some properties of a mesh: (a) different element types, (b) element shape and (c) a regular mesh.*

According to the Euler characteristic the average *valence of a vertex* is 6. In order to obtain similar valences for all vertices it is desired to have a valence close to 6 for all vertices in order to obtain a more *regular* mesh (see figure 3.3c).

Sharp edges and corners – known as features – usually should be preserved as they are essential for the general shape of a surface.

3.4. General Remeshing Process

The general remeshing process roughly follows the steps as outlined in Dunyach *et al.* (2013), Botsch *et al.* (2010), Botsch and Kobbelt (2004) and others.

First, a short description of the algorithm used for the uniform approach will be given; then, more details about the curvature-adaptive one will be provided. The remainder of this work will focus on techniques used for the implementation of the curvature-adaptive approach (Dunyach *et al.* 2013).

In the algorithm for uniform remeshing a target edge length L is defined, then repeatedly local mesh operations are performed. Edges are split into two edges if they are too long; too short edges are collapsed. In order to change vertex valences edge-flip is used. Vertices are shifted using tangential relaxation for a more regular mesh and they are projected back onto the original surface. This procedure is iterated about 5 to 10 times until edges approximately reach the desired edge length and the vertex valences are more equal.

This is the general algorithm:

```
remesh():
  # remeshing
  for (int i = 0; i < numberOfIterations; ++i):
    # control edge lengths
    for each edge in edges:
      if (edgeLength(edge) > 4/3 * L):
        split(edge)
      if (edgeLength(edge) < 4/5 * L):
        collapse(edge)

    # control vertex valences
    for each edge in edges:
      flip(edge)

    # optimise vertex positions
    for each vertex in vertices:
      tangentialRelaxation(vertex)

    # project vertices back onto surface
    for each vertex in vertices:
      backProjection(vertex)
```

The values 4/5 and 4/3 are chosen in Botsch and Kobbelt (2004) to have edges converging to uniform edge lengths.

In the main reference of this project, Dunyach *et al.* (2013), the values 4/5 and 4/3 were interchanged. As on the one hand it is not that useful to split edges that are already smaller than their target length and collapse edges that are too long and on the other hand other literature (e.g. Botsch and Kobbelt (2004), Botsch *et al.* (2010)) use these values differently the version stated above was used.

With having the goal to get a mesh with curvature-adaptive sampling and feature preservation as in Dunyach *et al.* (2013), the general process to optimise the mesh is extended: First, the curvature of the mesh is calculated and features are determined. Next, the edge lengths of the mesh are adapted using local operations as described in the next section. The desired edge length is obtained from the curvature: Instead of a fixed target edge length L a local edge length $L(e)$ is used. The relocation of vertices is also done related to this local target edge length. For feature preservation some restrictions on the general mesh operations are added.

The next sections offer more detail on these operations, important conditions for them and the handling of curvature and features.

3.5. Local Mesh Operations

With local mesh operations a mesh can be optimised to become a more regular mesh. Operations such as edge-split, edge-collapse and edge-flip create a mesh with edge lengths and valences that are more equal; tangential relaxation and back-projection relocate vertices. These local mesh modification operations can be seen in figure 3.4. Boundary edges and feature edges need a special handling as for the first ones less information is available and for the latter ones the shape should be preserved. Additionally, it has to be considered that the topology of the mesh in general should not be destroyed. Hence some special cases have to be added. For specific information on each operation see the following sections. The exact modifications performed in the edge-split, edge-flip and edge-collapse operations highly depend on the data structure (see chapter 4.3 for more details for changes required for the half-edge data structure).

3.5.1. Edge-Split

With edge-split (figure 3.4a) large edges can be split into two smaller ones at their midpoints. An edge is split supposed it is larger than a specific length. The new vertex position is calculated by:

```
newPoint = edge.leftPos + (edge.leftPos + edge.rightPos) / 2
```

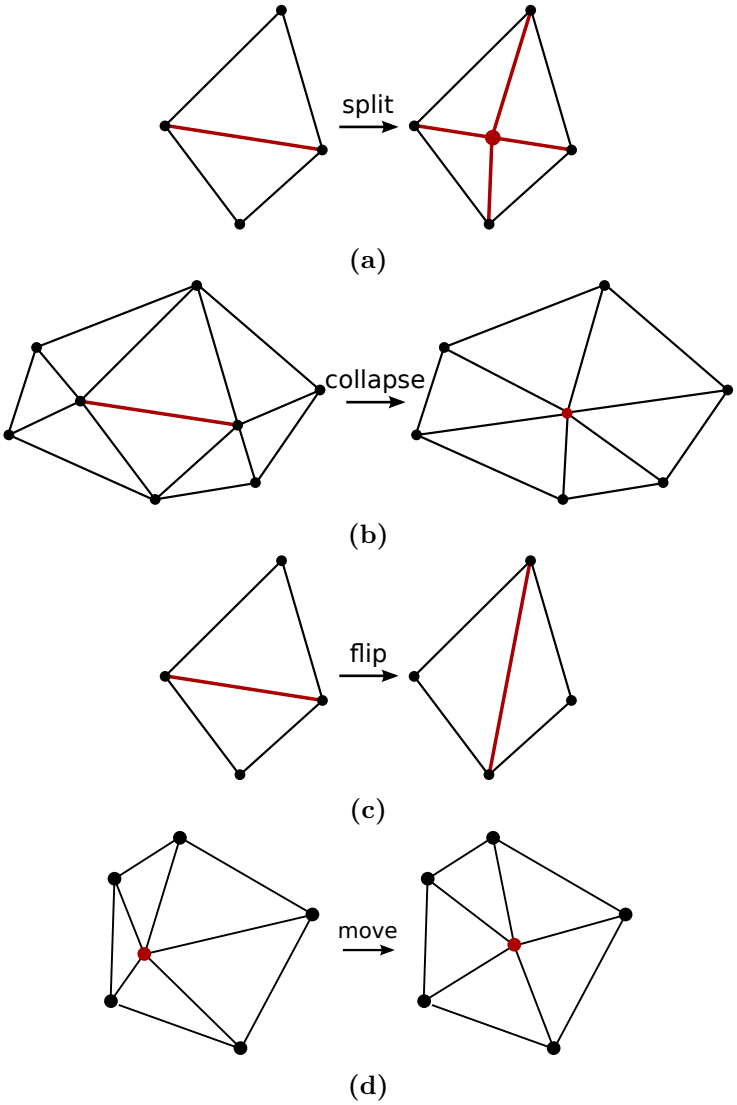


Figure 3.4.: *Local operations for a triangle mesh: (a) edge-split, (b) edge-collapse, (c) edge-flip and (d) vertex move.*

The result is that one new vertex is added to the mesh as well as three new edges and two triangles.

3.5.2. Edge-Collapse

Edges can be removed from the mesh by using edge-collapse (figure 3.4b). In the remeshing process this is done to remove too small edges.

This step can be imagined as removing an edge with two vertices and replacing it by a new vertex between the two previous vertices. The position is calculated in the same way as for edge-split. All edges connected to the previous vertices are connected to the new vertex.

3.5.3. Edge-Flip

Edge-flipping (figure 3.4c) is used to equalise the vertex valences. Usually valences of 6 and 4 are intended for interior and boundary vertices, respectively. The algorithm flips each edge and controls whether the deviation to the target valences decreases – i.e. whether the operation might bring an improvement to vertex valences. Supposed there will be no improvement the edge will not be flipped. An edge is flipped by changing the vertices of the edge to the neighbouring vertices in the adjacent triangles. If edge-flip would be used twice for an edge the previous topology could be restored.

3.5.4. Conditions

These previously described mesh operations do not always preserve the topology and could create non-manifold or unpleasant results. While edge-split can always be performed, edge-collapse and edge-flip need some tests before performing them to preserve the topological type and create satisfying results.

The following conditions have to be verified before using a mesh operation for an edge $e = \{\mathbf{v}_o, \mathbf{v}_1\}$; these conditions can be found in Hoppe *et al.* (1993), Dey *et al.* (1999) and Botsch *et al.* (2010).

First, according to Botsch *et al.* (2010) too large edges might be created by collapsing along chains of short edges, which would undo the work done by the split operation. Therefore, before collapsing it has to be checked that the newly-created edges will not be larger than the value used as threshold for splitting.

Next, for *collapse*, it should be prevented that the mesh intersects with itself (see figure 3.5b). To check this for the whole mesh would be too expensive. Hoppe *et al.* (1993) suggest to compare the maximum dihedral angle of edges incident to the new vertex after the operation. If this angle would be larger than a threshold angle the operation is rejected.

And finally, for preventing non-manifold results, the often so-called *link condition* has to be satisfied:

- If vertices $\mathbf{v}_0$ and $\mathbf{v}_1$ are boundary vertices the edge $e = \{\mathbf{v}_0, \mathbf{v}_1\}$ has to be a boundary edge, too (figure 3.6 shows the non-manifold result that would be obtained otherwise).
- For each vertex $\mathbf{v}$ incident to both vertices $\mathbf{v}_0$ and $\mathbf{v}_1$ there has to be a triangle $t = \{\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}\}$ (figure 3.5a shows the result that would be created otherwise).

The last condition can be expressed in another way:
The number of vertices that are adjacent to both $\mathbf{v}_0$ and $\mathbf{v}_1$ has to be exactly two: $|\mathcal{N}(\mathbf{v}_0) \cap \mathcal{N}(\mathbf{v}_1)| = 2$. If the edge is a boundary edge it has to be one. Here, $\mathcal{N}(\mathbf{v}_i)$ is the set of vertices adjacent to vertex $\mathbf{v}_i$.

For the *flip* operation the mesh has to be checked for self-intersection similar to the collapse operation. Figure 3.7 shows two examples for such invalid operations: in the first example a triangle would be created that has no surface area and hence no valid normal and in the second one the flipped edge creates two triangles with their normals roughly pointing into opposite directions. To prevent such an operation the dihedral angle of the edge after the operation is computed. If this angle is larger than a threshold value this might indicate that a new triangle has the wrong orientation.

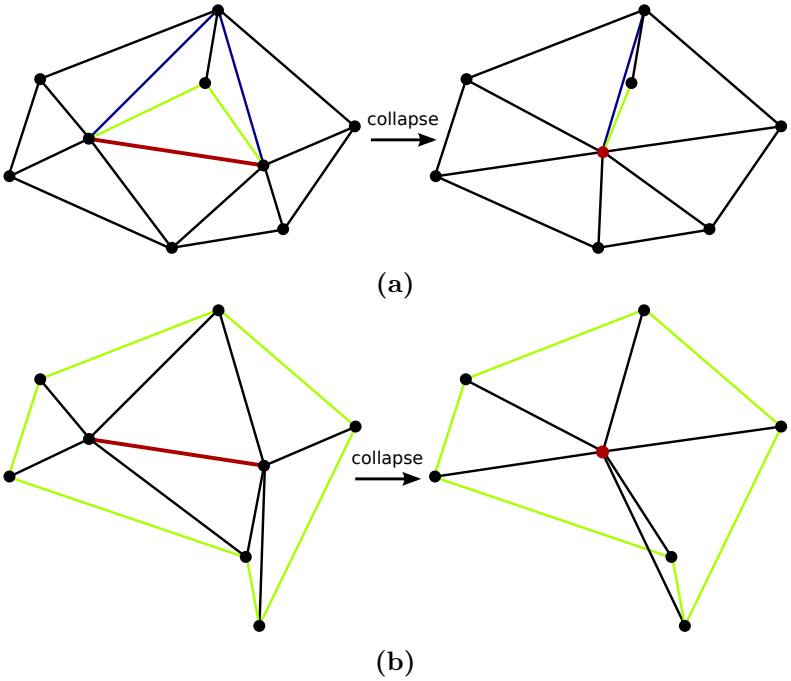


Figure 3.5.: *Meshes before (left) and after (right) edge-collapse for invalid scenarios: In (a) the result is non-manifold, in (b) a triangle changes its orientation.*

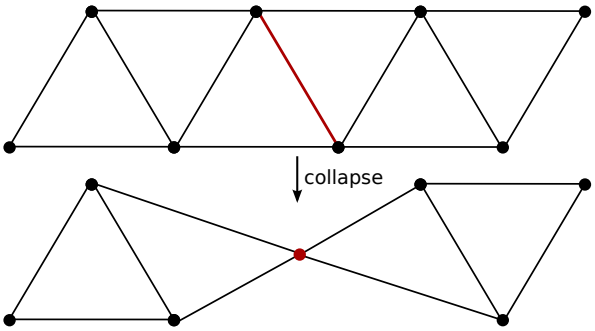


Figure 3.6.: *Meshes before (top) and after (bottom) edge-collapse for an invalid scenario: an edge with two adjacent boundary vertices that are not connected by a boundary edge create a non-manifold mesh.*

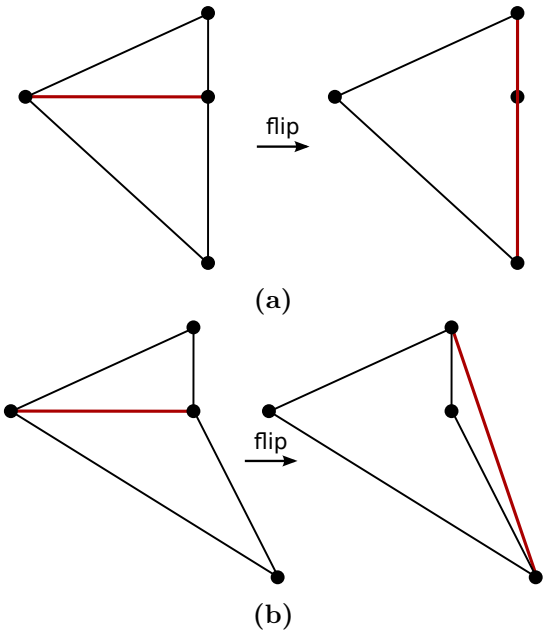


Figure 3.7.: *Meshes before (left) and after (right) edge-flip for invalid scenarios: (a) triangle with zero area would be created and (b) the new edge is outside the surface.*

3.5.5. Tangential Relaxation

For relaxation vertices are moved (figure 3.4d) in order to improve the vertex distribution. It can be used as an iterative smoothing filter on the meshes. Due to the back-projection to the surface afterwards, the movement is constrained to the vertex's tangent plane for stabilisation purposes.

For uniform remeshing each vertex $\mathbf{v}_i$ is moved to the average $\mathbf{c}_i$ of its one-ring neighbourhood $\mathcal{N}(\mathbf{v}_i)$ (Botsch and Kobbelt 2004):

$$\mathbf{c}_i = \frac{\sum_{\mathbf{v} \in \mathcal{N}(\mathbf{v}_i)} \mathbf{v}}{|\mathcal{N}(\mathbf{v}_i)|}$$

This is adapted for the curvature-adaptive approach (Dunyach *et al.* 2013) as the target edge lengths are not uniform and the local target edge sizes have to be preserved. Average triangle barycentres are used to compute an *Optimal Delaunay Triangulation* (Chen and Holst 2011). $\mathbf{c}_i$ is computed as average of the barycentres $\mathbf{b}_j$ of the triangles $t_j \in \mathcal{T}_i$ incident to the vertex, weighted by the surface area of the triangles and the sizing field at $\mathbf{b}_j$ (the sizing field

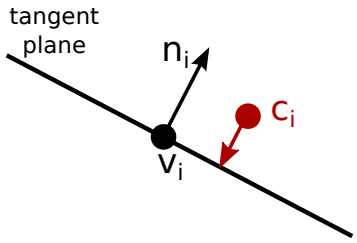


Figure 3.8.: *Projection to tangent plane.*

defines the local target edge length; see chapter 3.7):

$$\mathbf{c}_i = \frac{\sum_{j \in \mathcal{T}_i} |t_j| L(\mathbf{b}_j) \mathbf{b}_j}{\sum_{j \in \mathcal{T}_i} |t_j| L(\mathbf{b}_j)}$$

This method was chosen for reasons of robustness and simplicity. Another approach could use circumcentres instead of barycentres.

Finally, the position is updated by projecting it into the tangent plane $(\mathbf{v}_i, \mathbf{n}_i)$ as visualised in figure 3.8:

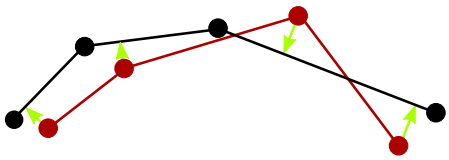
$$\mathbf{v}_i = \mathbf{c}_i + \mathbf{n}_i \mathbf{n}_i{}^{\mathrm{T}} (\mathbf{v}_i - \mathbf{c}_i)$$

According to Botsch *et al.* (2010) this can be implemented as:

$$\mathbf{v}_i = \mathbf{c}_i + (\mathbf{n}_i \times (\mathbf{v}_i - \mathbf{c}_i)) \mathbf{n}_i.$$

3.5.6. Projection to Surface

The vertices can be projected to the nearest point of the reference mesh. This step is mentioned as optionally in Dunyach *et al.* (2013).



(a)

Figure 3.9.: *Back-projection of new vertex positions onto the reference mesh – the original mesh is visualised in black, the new vertex positions in red, the green arrows show where they are moved to.*

For each vertex the nearest position on the original surface has to be found. A naive way to do this is by iterating over each triangle of the reference mesh for each vertex in order to find this position. For each vertex and triangle the minimum distance and the corresponding position is calculated. The position of the triangle with minimum distance is used. Unfortunately, this is quite expensive as many triangles are considered even if they are far away from the position.

A better solution would be the precalculation of a kD-tree for the triangles of the initial mesh during the initialisation. Then, the nearest triangles to a position can be found efficiently and the closest point can be detected by calculating the distance to these triangles only.

For the calculation of the position closest to a triangle Eberly (1999) describes an effective algorithm for a calculation in three-dimensional space.

Finally, the position of the vertices can be moved to the detected position on the reference mesh (see figure 3.9).

As this step was considered as optional only the naive implementation was realised in this project to compare the resulting meshes to the results without back-projection.

3.6. Boundary Handling

The implemented data structure supports meshes with a boundary. In order to handle these boundaries correctly and to preserve the topology of the mesh adaptations to the basic mesh operations (edge-split, collapse, flip and tangential relaxation) have to be taken into account.

Edge-split is performed for one triangle only as no second triangle is available. Instead of replacing two triangles by four triangles, one triangle is replaced by two.

Edge-collapse works in the same way for one triangle and the vertices adjacent to the edge. Some conditions related to boundaries that are necessary for preserving the topological type were already stated in section 3.5.4.

Edge-flip is not possible to be performed on boundaries as it needs exactly two triangles to flip an edge.

For *tangential relaxation* the boundary should not be destroyed. Vertices can be moved along boundary edges only.

3.7. Target Edge Length

Curvature Calculation

The curvature of the surface is used for calculating a sizing field to determine edge lengths. Based on this the curvature-adaptive remeshing can be carried out.

Actually, triangle meshes have no curvature as all faces are flat. It is possible to think of a mesh as a piecewise linear approximation of an unknown surface. The challenge is to estimate the curvature of this unknown surface. A survey of curvature calculation techniques can be found in Petitjean (2002). The current standard is the calculation of a mean curvature H and Gaussian curvature K; from these curvatures a maximum absolute curvature κ can be calculated. Such methods are described for instance in Dyn *et al.* (2001) or Meyer *et al.* (2003). The implemented curvature calculation is based on Dyn *et al.* (2001).

Below, formulas are given to calculate a curvature value for a vertex $\mathbf{v}$. n is the number of vertices in the one-ring neighbourhood of the vertex $\mathbf{v}$; A is an area attributed to $\mathbf{v}$; α_i are the incident triangle angles around $\mathbf{v}$; β_i are the dihedral angles between triangle t_{i-1} and t_i that are incident to the edge $e_i = \{\mathbf{v}, \mathbf{v}_i\}$ and θ_{i-1} and θ_{i+1} are the two angles opposite to the edge e_i in the triangles incident to this edge. The relationship between these variables can be seen in figure 3.10.

The curvature of the mesh is calculated for each vertex $\mathbf{v}$ in the following way:

The mean curvature is calculated by:

$$H = \frac{1}{4A} \sum_{i=0}^{n} ||e_i||\,|\beta_i|$$

and the Gaussian curvature by:

$$K = \frac{1}{A} \cdot (2\pi - \sum_{i=0}^{n} \alpha_i)$$

The maximum absolute curvature is:

$$\kappa = \begin{cases} |H| & K \leq 0, \\ \sqrt{|H|^2 - K} & \text{otherwise.} \end{cases}$$

For the surface area the *Voronoi area* is often used:

$$A = \frac{1}{8} \sum_{i=0}^{n} (cot\,\theta_{i-1} + cot\,\theta_{i+1} ||\mathbf{v} - \mathbf{v}_i||^2)$$

A simpler way is to use the *barycentric area*, the most commonly used area in literature according to Dyn *et al.* (2001):

$$A = \frac{1}{3} \sum_{j \in T_i} |t_j|$$

This is one third of the area of all triangles t_j adjacent to $\mathbf{v}$.

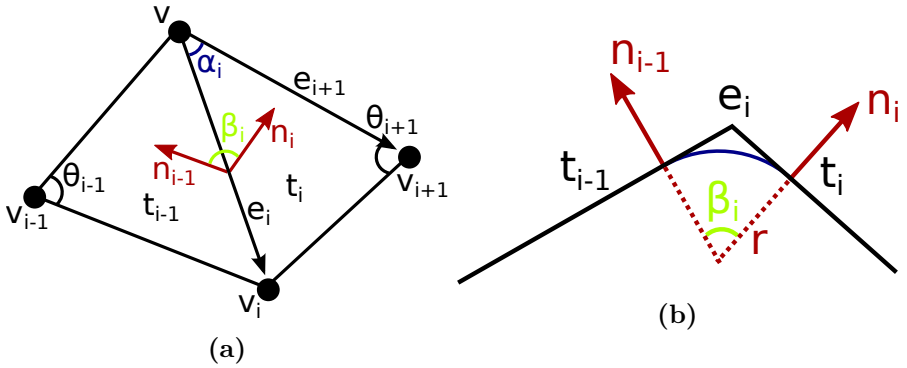


Figure 3.10.: *Relationship between variables required for curvature calculation: (a) top view and (b) side view*

For the calculation of the curvature in this project a further parameter γ is added to perform a scaling of the values calculated for κ.

$$\bar{\kappa} = (\frac{\kappa_i}{\kappa_{max}})^{\gamma} \cdot \kappa_{max}$$

Here, the curvature value is scaled to values from 0 to 1 by dividing by the maximum curvature value κ_{max} of all vertices. Then, the value is raised to the power of γ. Finally, the value is scaled back to the previous codomain. If γ is smaller than 0 more vertices have a higher curvature value, i.e. there will be more small triangles. If γ is larger than 0 there will be more larger triangles as the curvature value gets smaller.

The curvature values for each vertex can be visualised using a colour value for the minimum and maximum curvature values; the values in between are interpolated. Within triangles the colour values are interpolated using the incident vertices. Figure 3.11 shows the calculated curvature values for an example mesh: White is used for the minimum curvature and red for maximum curvature as seen in the corresponding legend.

In Vorsatz *et al.* (2001) the curvature is smoothed in order to remove noise. The disadvantage is that smaller features might be missed and a higher computation time is required.

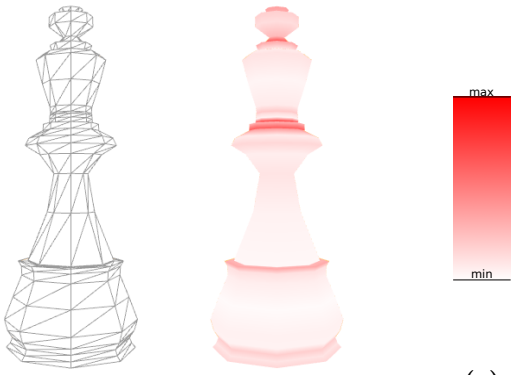


Figure 3.11.: *Curvature calculation for an example mesh: (a) Reference mesh, (b) mesh colour coded by curvature (high curvature: red; low curvature: white) and (c) corresponding legend.*

Sizing Field

The main difference between the curvature-adaptive approach Dunyach *et al.* (2013) and the uniform approach Botsch and Kobbelt (2004) is that the constant target edge length L is replaced by an adaptive sizing field $L(\mathbf{v})$.

This sizing field is related to the curvature of the mesh and an additional parameter, the approximation tolerance ϵ. This tolerance is a threshold for the allowed geometric deviation of the triangle mesh from the underlying smooth surface.

The sizing value for each vertex can be computed from the curvature value $\bar{\kappa}_i$ and the approximation tolerance ϵ:

$$L(\mathbf{v}_i) = \sqrt{\frac{6\epsilon}{\bar{\kappa}_i} - 3\epsilon^2}$$

The relationship between this formula for the target edge lengths and the curvature of the mesh is explained in detail in Dunyach *et al.* (2013).

The values for the sizing field can be clamped to user defined bounds: $[L_{min}, L_{max}]$. The sizing value $L(e)$ for an edge $e = \{\mathbf{v}_0, \mathbf{v}_1\}$ is determined as the minimum of the sizing values for both endpoints $\mathbf{v}_0$ and $\mathbf{v}_1$: $L(e) = min\{L(\mathbf{v}_0), L(\mathbf{v}_1)\}$.

The sizing field for a barycentre of a triangle is calculated as the average of the sizing field of the triangle vertices.

3.8. Feature Detection and Preservation

Many surfaces contain sharp corners, edges or material boundaries which should be preserved when creating a new mesh as they are significant for the general shape – these elements are generally called features.

An example could be a general cube. If the edges were not handled as features the general shape would be smoothed and might result in a sphere. With feature preservation specific edges are changed in a limited way only. Features have to be defined on the original mesh before remeshing takes place. This can be done algorithmically or by interactive specification. The approach used here uses an algorithmic detection and furthermore some rules are added to the previously described topological and geometrical mesh operations similar to Vorsatz *et al.* (2003).

Feature Detection

To detect features for each edge the angle α between incident faces is checked (see figure 3.12). Usually the dihedral angle β is calculated for this purpose. β is the angle between the two normal vectors $\mathbf{n}_0$ and $\mathbf{n}_1$ of the triangles t_0 and t_1 incident to an edge. The relation between α and β can be seen in figure 3.12b: $\beta = 180° - \alpha$. If the dihedral angle β is larger than a specified threshold angle β_t the edge is defined as a feature edge and can be preserved in the further remeshing process.

The dihedral angle can be calculated by using the normal unit vectors $\mathbf{n}_0$ and $\mathbf{n}_1$ of the triangles t_0 and t_1 adjacent to the edge e_1 using the dot product as follows:

$$cos(\beta) = |\mathbf{n}_0 \cdot \mathbf{n}_1|$$

Figure 3.13 shows some results for feature detection using different threshold angles β_t; β_t is increased from left to right.

This approach to detect features was used by Dunyach *et al.* (2013), Fu and Zhou (2009) and Alliez *et al.* (2002), just to mention a few. In Lévy *et al.* (2002) this method was slightly extended to filter noise. Different and more complex approaches exist for this task.

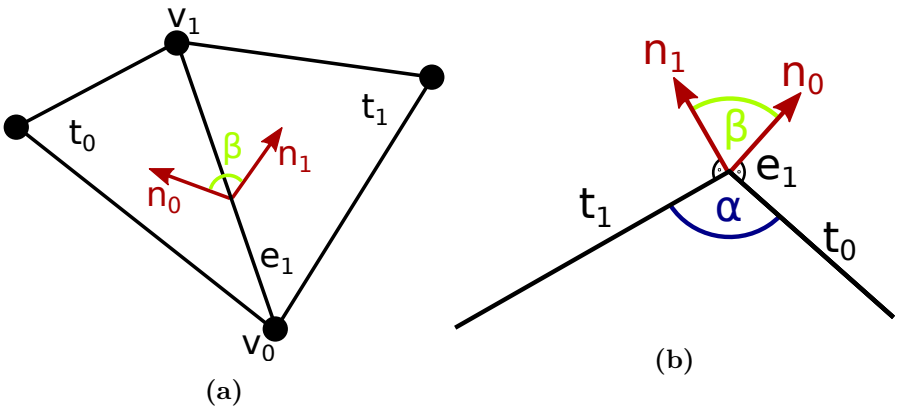


Figure 3.12.: *Calculation of dihedral angle β between triangle t_0 and t_1. (a) Top view of the relation between the angle β and the triangles incident to edge e_1 and (b) side view for the relation between angle α and β.*

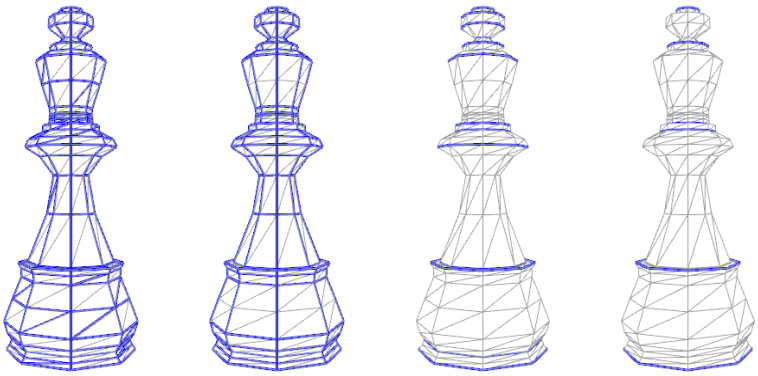


Figure 3.13.: *Feature extraction with different threshold angles for an example mesh: (a) $\beta_t = 0$, (b) $\beta_t = 10$, (c) $\beta_t = 50$ and (d) $\beta_t = 100$.*

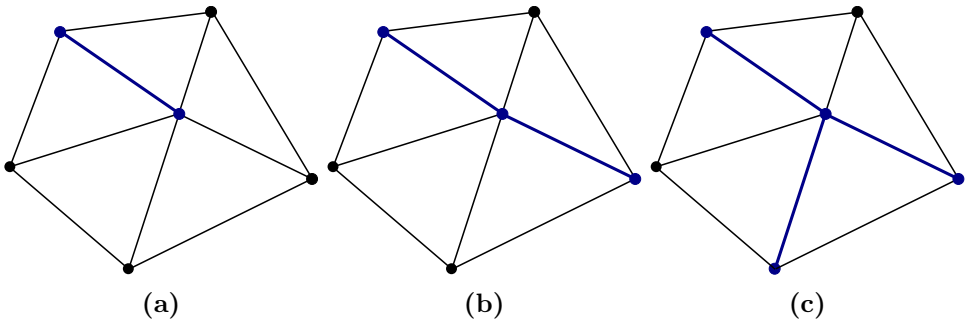


Figure 3.14.: *Examples for feature edges and vertices (highlighted in blue): (a) one feature edge, (b) two feature edges (feature line) and (c) more than two feature edges (corner vertex).*

Feature Preservation

Features need special consideration during the used meshing operations in order to be preserved.

Generally, edge-splits, edge-collapses and edge-flips that would destroy feature edges are not allowed (Botsch *et al.* (2010), Vorsatz *et al.* (2003)); this leads to:

- *Corner vertices* are vertices with more than two incident feature edges or exactly one (figure 3.14a and 3.14c). They can not be moved and have to be preserved; they are not used for geometric and topological operations.
- *Feature vertices* are vertices with two incident feature edges (figure 3.14b). They move along feature lines only.

Especially for the preservation of features for adaptive remeshing:

- The sizing values for feature vertices are computed in a different way to avoid high sampling densities near feature edges. The values are computed as the average of their non-feature neighbours.
- Tangential relaxation is performed along the feature lines only.

In practice, this means the following for the mesh operations:

Edge-split can be performed in the usual way, while it needs to be considered that the new vertex and the new edges that replace the feature edge are marked as feature elements.

For *edge-collapse* corner vertices can not be moved; the operation is only allowed for feature edges if both incident vertices are feature vertices but not corner vertices.

Feature edges can not be *flipped* as they would be destroyed throughout this operation.

During *tangential relaxation* vertices can be moved along boundary edges only. For the calculation the incident triangles are replaced by the two incident feature edges, triangle areas by edge lengths and barycentres by midpoints.

3.9. Triangle Calculations

Vector operations are used to modify the mesh. These include common operations such as the calculation of the cross product, scalar product or the length of a vector. Some operations necessary for the remeshing algorithm are described below.

Surface Area

For some mesh operation it is required to know the surface area of the triangles. There are multiple ways to calculate the area of triangles. Depending on the available information it can be calculated using different methods.

The best-known formula is $A = \frac{1}{2}bh$ with b being the length of a side – usually known as the base – and h the length of the line perpendicular to the base from the vertex opposite the base to the base (usually known as the height).

As in our case all positions of the vertices are known the calculation is based on the calculation of the area of the parallelogram defined by the vectors for the edges $\bar{\mathbf{e}}_0 = \mathbf{v}_0 - \mathbf{v}_1$ and $\bar{\mathbf{e}}_1 = \mathbf{v}_2 - \mathbf{v}_1$ as visible in figure 3.15.

The area of the parallelogram is calculated using the cross product:

$$A = |\bar{\mathbf{e}}_0 \times \bar{\mathbf{e}}_1|.$$

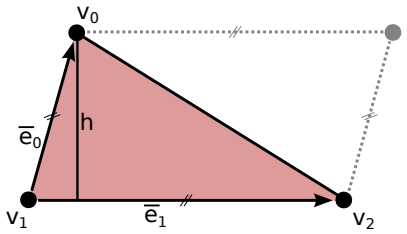


Figure 3.15.: *Surface area of an arbitrary triangle.*

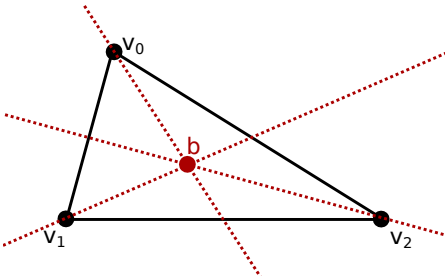


Figure 3.16.: *Barycentre of an arbitrary triangle.*

Hence, the area of the triangle is:

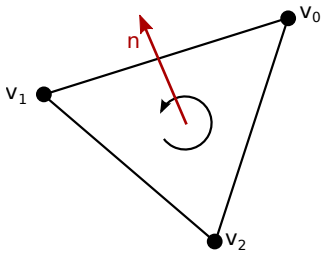
$$A = \frac{1}{2}|\bar{\mathbf{e}}_0 \times \bar{\mathbf{e}}_1|$$

Barycentre

For the calculation of the tangential relaxation (see chapter 3.5.5) the barycentres of triangles are required. In mathematics and physics usually the term centroid or geometric centre is used.

The barycentre of a triangle is the position of intersection of its medians (figure 3.16). A median is the line from the midpoint of a line to the opposite vertex. Each of the medians is divided in the ratio 1:2 by the barycentre. The coordinate of the barycentre is the mean of the three vertices' positions:

$$b = \frac{\mathbf{v}_0 + \mathbf{v}_1 + \mathbf{v}_2}{3}$$



(a)

Figure 3.17.: *Triangle and its normal; vertices are oriented anticlockwise in the triangle.*

Angle Between Vectors

Especially for feature detection and curvature calculation the angle between two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ has to be calculated. This can be done using the scalar product:

$$\alpha = acos(\frac{\mathbf{v}_0 \cdot \mathbf{v}_1}{|\mathbf{v}_0||\mathbf{v}_1|})$$

Here the smaller angle between these vectors is determined.

Triangle and Vertex Normals

Normals of triangles and vertices are important for triangle meshes.

For a triangle the vertices $\mathbf{v}_0, \mathbf{v}_1$ and $\mathbf{v}_2$ need to be ordered in anticlockwise direction to obtain the correct result (figure 3.17). Using the cross product for the edges $\bar{\mathbf{e}}_0 = \mathbf{v}_0 - \mathbf{v}_1$ and $\bar{\mathbf{e}}_1 = \mathbf{v}_2 - \mathbf{v}_1$ the normal can be calculated:

$$\mathbf{n}_t = \bar{\mathbf{e}}_0 \times \bar{\mathbf{e}}_1$$

For each vertex the normal is calculated as weighted average of the normals of the triangles adjacent to it. The weights are proportional to the angles of the corresponding triangles at the vertex. Finally the normal is normalised. This is similar as in Surazhsky and Gotsman (2003).

A simpler version would be the calculation by averaging the normals of all adjacent triangles.

4. Implementation

In this chapter the focus is on implementation details concerning the application in general, the internal data structure, supported mesh formats and preprocessing steps for a given input mesh. The general remeshing process used for the implementation has similar steps as described in the publication Dunyach *et al.* (2013) and was outlined in the previous chapter.

4.1. Workflow

The general workflow when using the application as visualised in figure 4.1 is as follows:

First, a mesh can be loaded. Oriented manifold meshes, optionally with boundary, of arbitrary genus and with triangles and quadrangles can be imported. The mesh needs to be prepared to be used by the application: It is processed to consist of triangles only, duplicated vertices are eliminated, triangles containing a vertex more than once or with no surface area are removed. Then, the mesh is converted to the half-edge data structure. This structure allows access to neighbouring elements of vertices, edges and triangles in order to perform local modifications on the mesh. Normals of triangles and faces are calculated. Settings for the calculation of the sizing field (approximation tolerance or target edge length) and for feature detection (dihedral angle) can be adjusted in the user interface.

Next, the curvature of the mesh is calculated to determine the sizing field and feature edges are extracted. Mesh operations (edge-split, edge-collapse, edge-flip as well as tangential relaxation and optionally back-projection) are performed in order to create a better mesh (curvature-adaptive or uniform) for the surface while preserving features and hence the general shape. These mesh operations are iterated several times.

Finally, the mesh is visualised. The result can be exported as an image or in a mesh format. Additionally it is possible to export current settings from the user interface.

This is the workflow in graphical form:

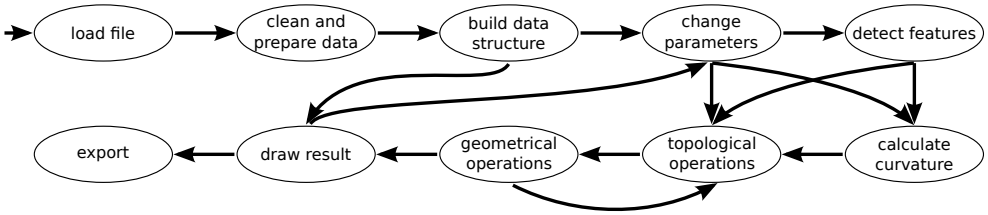


Figure 4.1.: *Workflow of application usage.*

The next sections focus on details for the realisation of this process. The data structure used will be described; supported data formats and the mesh preparation steps, how the software system is designed and some features of the user interface follow.

4.2. Mesh Import and Export

Files with mesh information can be imported to visualise and remesh the contained mesh. Results obtained by remeshing can also be saved and imported in other software packages.

The application is able to handle OFF, OBJ and PLY files for importing and exporting meshes (only ASCII files are supported). Depending on the file extension a specific mesh loader or exporter is started. For importing, data for vertex positions and faces is extracted; for exporting data is written to a file. The importers use the Boost library (Spirit.Qi) for parsing the files.

Each of the file formats specifies a mesh by vertices and faces using indices to shared vertices. Both quadrangles and triangles are allowed for faces (quadrangles are triangulated for further processing, see section 4.4). Additional information (as for instance vertex normals) can be present in the files but the information is not used. The structure of the code is designed to easily add further parsers and exporters. Next, there will be more detail for the supported files. Example files for a cube can be found in appendices A-C.

4.2.1. Object File Format

The Object File Format (OFF) is a simple geometry format defining a mesh by a list of vertices and faces.

This is the structure of a supported OFF file:

```
OFF
# This is a comment
# Number of vertices, faces and edges
NV NF NE
# Vertices: x, y and z coordinates
x[0] y[0] z[0]
⋮
x[NV - 1] y[NV - 1] z[NV - 1]
# Faces: Number of vertices of face, indices to vertices
Nv[0] v[0,0] v[0,1] ··· v[0, Nv[0] - 1]
⋮
Nv[NF - 1] v[NF - 1, 0] v[NF - 1, 1] ··· v[NF - 1, Nv[NF - 1] - 1]
# Further elements might follow ···
```

The file usually starts with the file signature OFF. However, this is only optional. Next there has to be a line consisting of three integers: the number of vertices N_V, the number of faces N_F and the number of edges N_E. As the number of edges usually is not used – this is the case in this implementation, too – the value may be 0 but has to be available. In the next lines the vertex coordinates are specified. The number of lines for vertices are specified in the header. Each line contains information for one vertex. This implementation uses the first three float values for the x, y and z values. Further values for normals can be included but are ignored. Finally the faces are defined. The first value indicates the number of edges of the face – this implementation supports three and four vertices. Next, indices to the previously defined vertices define the face. Note that the vertices are indexed starting with index 0. All empty lines and additional white space are ignored. All lines starting with a hash sign (#) or parts of a line after a hash sign are ignored. The hash sign can thus be used for comments.

Special types of the OFF format exist (COFF, NOFF, $\cdots$) – this might result in some files not being parsable. All common OFF files are supported.

4.2.2. PLY Format

The PLY format (Polygon File Format) was initially used for three-dimensional data from 3D scanners.

The typical structure of a PLY file is: **Header** → **Vertices** → **Faces** (→ Lists of other elements).

The header specifies the format of the file (binary or ASCII) and provides information about the data listed later. The supported file format is ASCII. As it is only of interest to extract vertex and face data further specifications are ignored. The structure of a header looks like this:

```
ply
format ascii 1.0
comment This is a comment
element vertex NV
property float x
property float y
property float z
element face NF
property list uchar int vertex_indices
end_header
```

Here, N_V and N_F specify the number of vertices and faces that follow in the file, respectively.

The first line has to contain the keyword *ply* that identifies the file as a PLY file. The next line specifies whether the file contains ASCII or binary data. Comments follow the keyword *comment* at the beginning of a line and are ignored.

The keyword *element* is used to specify for vertices, faces and optionally other elements how they are stored and how many of these elements will be listed. Specific data for these elements is indicated by the keyword *property*. For vertices this includes x, y and z coordinates; additionally normal or colour information could be added. Vertices and faces have to be the first two element types. It is possible to add further element types but they are ignored.

The body of the file is simply a list of vertices and faces with references to the vertices. Vertices are specified by x, y and z coordinates. Indices start at 0. For each face the number of vertices is specified first followed by indices.

This is the structure of the data containing vertices and faces:

```
x[0] y[0] z[0]
⋮
x[NV - 1] y[NV - 1] z[NV - 1]

Nv[0]  v[0,0] v[0,1] ··· v[0, Nv[0]- 1]
⋮
Nv[NF - 1]  v[NF - 1, 0] v[NF - 1, 1] ··· v[NF - 1, Nv[NF - 1] - 1]
```

4.2.3. OBJ Format

OBJ is a common format used for computer animation and in modelling software. Only a subset of the data available in an OBJ file is used. The vertex and face data is important.

Below, there is the layout for supported OBJ files:

```
# This is a comment
# Vertices: v followed by x, y and z coordinates
v  x[1] y[1] z[1]
v  x[2] y[2] z[2]
⋮
# Normals: vn followed by nx, ny and nz
vn  nx[1] ny[1] nz[1]
vn  nx[2] ny[2] nz[2]
⋮
# Texture coordinates: vt followed by u, v (and w)
vt  u[1] v[1] w[1]
vt  u[2] v[2] w[2]
⋮
# Faces: f followed by indices to vertices
#        (and normals and triangles)
f  v[1,1]/vn[1,1]/vt[1,1] v[1,2]/vn[1,2]/vt[1,2] ···
f  v[2,1]/vn[2,1]/vt[2,1] v[2,2]/vn[2,2]/vt[2,2] ···
⋮
```

Vertices are given by the letter v followed by their x, y and z coordinates. Optionally, data for normals and texture coordinates can follow. Next, faces are defined: each face starts with the letter f followed by a list of vertex indices and optionally by normal and texture coordinate indices ($f\ v_1/vn_1/vt_1\ v_2/vn_2/vt_2\ \ldots$); listing the vertex indices ($f\ v_1\ v_2\ \cdots$) is enough for this application. Indices start at 1. Only the vertex index is used by the application. Each vertex and face is specified on a separate line.

Comments start with a hash sign and blank lines are ignored; lines that can not be used by the application (e.g. normal data or lines starting with unknown keywords) are ignored, too.

4.3. Mesh Data Structure

The selection of an appropriate data structure for the representation of the mesh is crucial for most geometric algorithms. An overview of some of the most common data structures can be found for example in Botsch *et al.* (2010) which helped with the decision for the data structure used in this project.

Both topological and algorithmical demands have to be considered when choosing a specific data structure. Some structures are suited for 2-manifold meshes only; others are limited to triangle meshes or have other limitations or improvements. For algorithmical purposes it is often necessary to have efficient access to the local neighbourhood of vertices, edges and faces and to perform modifications to the mesh. Sometimes it is sufficient if the data structure is efficient for simply rendering the mesh.

A simple structure for meshes is a list of polygonal faces with their three-dimensional vertex positions. This structure has no information about the mesh connectivity and vertices are replicated for each face they are a part of. By using indices to shared vertices (i.e. to a list of vertex positions that can be referenced by faces with the index in the list) the redundancy can be avoided. This structure is often used for mesh file formats such as OFF, PLY or OBJ (see chapter 4.2 for more details about these formats). It is also efficient for rendering using for example OpenGL Vertex Array Objects. Nevertheless, the main disadvantage of such a data structure is that access to the local neighbourhood of elements of the mesh (vertices and triangles) is expensive as the whole mesh has to be searched – the lookups take a lot of time. This is not convenient for most geometric algorithms as it is the case for this project, too.

4.3.1. Desired Properties

The algorithms for the remeshing approach require/are limited to the following data characteristics and access patterns:

- All faces are triangles
- Traversal over each vertex, edge and triangle
- Access to all vertices of a triangle in anticlockwise direction
- Access to incident triangles of an edge
- Access to both endpoints of an edge
- Access to all neighbouring vertices/triangles/edges of a vertex in (anti)clockwise order, i.e. all elements in the one-ring neighbourhood of a vertex

In general, it is required to have the possibility to traverse the whole mesh and to have access to the local neighbourhood of elements in order to perform mesh modifications.

Local operations based on edges (edge-split, edge-collapse and edge-flip) and the curvature calculation need a way to get access to the local neighbourhood to delete, add or change vertices and, hence, edges and triangles. The calculation of features needs information of triangles adjacent to edges.

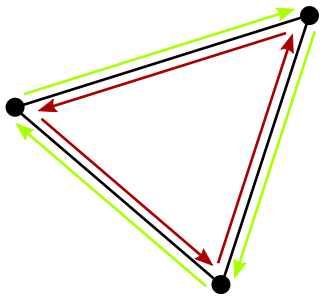


Figure 4.2.: *Half-edge data structure: Each edge is replaced by two half-edges pointing in opposite directions.*

For this project the so-called half-edge data structure was chosen as it seems to be appropriate for such needs. More memory is required but the performance is better. The data structure is convenient for both traversing the mesh as well as modifying it.

In the following the description is limited to triangles only as triangle meshes are used, although the general data structure can be used for other polygons, too.

4.3.2. Half-Edge Data Structure

The *half-edge data structure* is a common way of representing meshes when local modifications are required. The implemented structure is based on the description in Botsch *et al.* (2010). The structure is also known as *doubly connected edge list*. It is available in common mesh libraries such as *OpenMesh*[1] or *CGAL*[2].

This structure is edge-based and allows for each element of the mesh to access its neighbouring elements. It is suitable for orientable manifold meshes with boundary. The data structure consists of lists of vertices, triangles and half-edges. The main characteristic is that each edge is split into two oriented half-edges pointing in opposite directions as visible in figure 4.2 for a triangle. Half-edges are oriented in anticlockwise direction in triangles. Boundaries have no adjacent triangular face.

[1] www.openmesh.org
[2] www.cgal.org

The following references have to be stored for each element:

For each **vertex** (see figure 4.3a):

- Three-dimensional position: $p \in \mathbb{R}^3$
- Reference to an outgoing half-edge

Only the reference to one half-edge is needed; it does not matter which one.

For each **triangle** (see figure 4.3b):

- Reference to one of its half-edges (does not matter which one)

For each **half-edge** (see figure 4.3c):

- The vertex it points to
- Its incident triangle
- The next half-edge of the face in anticlockwise direction

A reference to the previous half-edge in the face could be stored, too. As only triangles are used storing the previous half-edge directly can be omitted and it can be accessed by using:

```
prevHalfEdge = nextHalfEdge(nextHalfEdge(halfEdgeIndex))
```

Two opposite half-edges are stored in the list subsequently and build pairs for each edge. The opposite half-edge can be accessed implicitly by:

```
oppositeHalfEdgeIndex = halfEdgeIndex -
                        2 * (halfEdgeIndex % 2) + 1
```

As two half-edges are grouped to pairs access to full edges is possible (e.g. if all edges shall be traversed instead of all half-edges).

The references are implemented using indices. Instead of indices pointers could be used. Indices are more flexible though and allow efficient memory relocation.

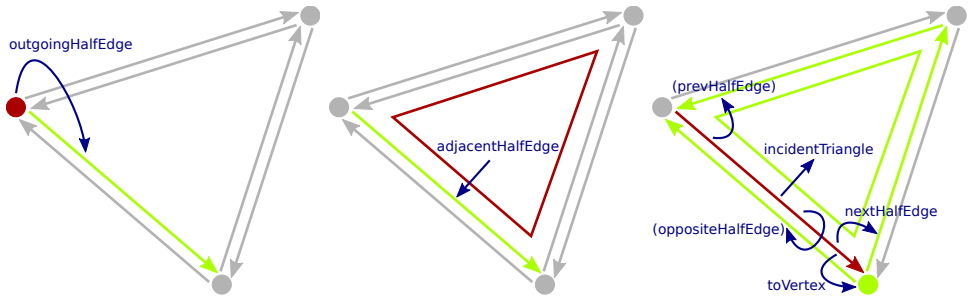


Figure 4.3.: *Referenced data in the half-edge data structure for (a) a vertex, (b) a triangle and (c) a half-edge.*

Boundaries

The basic data structure can handle manifold meshes. With small extensions it can also handle manifold meshes with a boundary.

Boundary edges have only one incident triangle. This means that one of the half-edges representing this edge contains information about the existing triangle; the opposite half-edge has no incident triangle and hence no next half-edge. The indices to these non-existent elements is set to -1 to detect the boundary. The opposite vertex can be accessed as usual as half-edges are grouped and the index of the vertices the half-edges are pointing to have to be valid, too. This extension has influence especially when elements of the one-ring-neighbourhood of a vertex are requested.

Indices to next half-edges along a boundary could be stored. Building the initial mesh would be too expensive though as for each boundary edge the next half-edge needs to be found to set the reference to the correct next half-edge.

Mesh Navigation

Below, there will be some examples showing how to access neighbouring elements for vertices, edges and triangles.

Elements in the one-ring neighbourhood of a vertex

Most important is the access to the one-ring neighbourhood of a vertex: vertices/triangles/outgoing half-edges. Supposed there are no boundaries this code could be used:

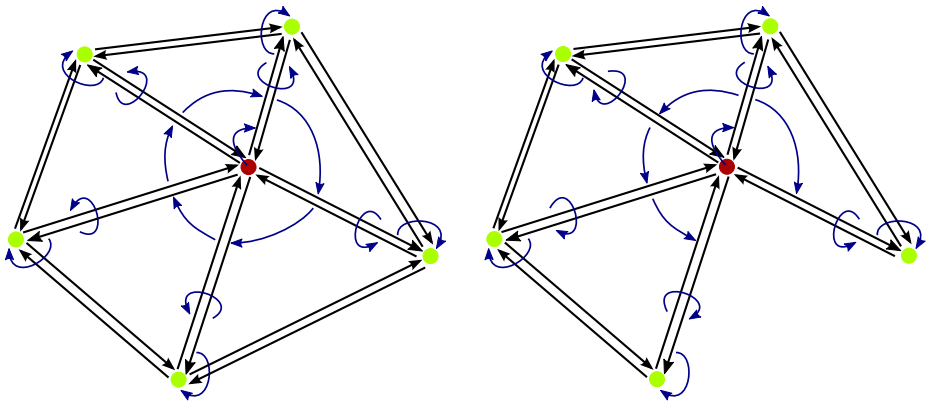
```
accessNeighbouringElements(int centredVertexIndex):
  int halfEdgeIndex = outgoingHalfEdge(centredVertexIndex)
  int halfEdgeStop = halfEdgeIndex
  do:
    # for vertices
    int vertexIndex = toVertex(halfEdgeIndex)
    # or for triangles
    int triangleIndex = incidentTriangle(halfEdgeIndex)

    # do something with vertexIndex/triangleIndex/halfEdgeIndex

    halfEdgeIndex = nextHalfEdge(oppositeHalfEdge(halfEdgeIndex))
  while (halfEdgeIndex != halfEdgeStop)
```

This function starts with an outgoing half-edge of the vertex. The remaining outgoing half-edges can be accessed in clockwise order by navigating to the next half-edge of the opposite half-edge. This is repeated until the first half-edge is reached again. Depending on the required data in the neighbourhood of a vertex the lines for *vertexIndex* and *triangleIndex* can be removed. Figure 4.4a shows this procedure for accessing vertices.

For meshes with boundaries this procedure has to be extended. As previously mentioned half-edges that represent a boundary have index -1 to their respective next half-edge. Instead of traversing this neighbourhood clockwise until the first half-edge is reached again elements are traversed clockwise until such an invalid element is reached and afterwards in the other direction to collect the remaining elements. See figure 4.4b for an illustration. In order to receive a list of elements in clockwise direction the first elements are appended while the latter elements are prepended to the list.



(a) (b)

Figure 4.4.: *Traversing the one-ring neighbourhood of a vertex to access all adjacent vertices (a) without boundary and (b) with boundary.*

This is the extended version of the code for accessing adjacent vertices:

```
accessIncidentHalfEdges(int centredVertexIndex):
  int halfEdgeIndex = outgoingHalfEdge(centredVertexIndex)
  int halfEdgeStop = halfEdgeIndex

  # traverse clockwise
  if (halfEdgeIndex != -1):
    do:
      # for vertices
      int vertexIndex = toVertex(halfEdgeIndex)
      # do something with vertexIndex
      halfEdgeIndex = nextHalfEdge(oppositeHalfEdge(halfEdgeIndex))
    while (halfEdgeIndex != halfEdgeStop && halfEdgeIndex != -1)

  # traverse anticlockwise
  if (halfEdgeIndex == -1):
    int prevHalfEdgeIndex = prevHalfEdge(
                            outgoingHalfEdge(centredVertexIndex))
    while (prevHalfEdgeIndex != -1):
      halfEdgeIndex = oppositeHalfEdge(prevHalfEdgeIndex)
      # for vertices
      int vertexIndex = toVertex(halfEdgeIndex)
      # do something with vertexIndex
      prevHalfEdgeIndex = prevHalfEdge(halfEdgeIndex);
```

Vertices of an edge

Incident vertices of an edge can be accessed as shown in figure 4.5a:

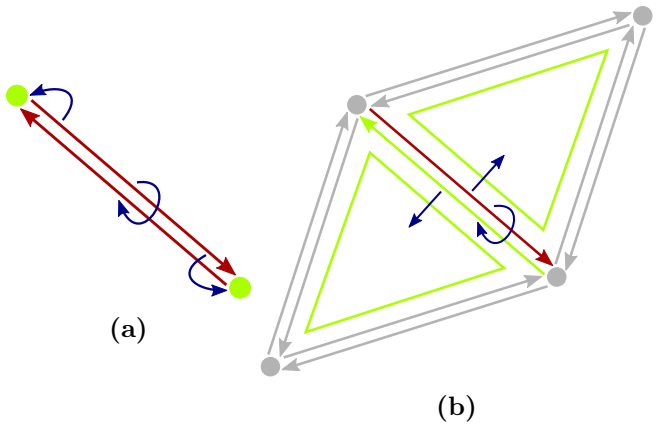


Figure 4.5.: *Access to neighbouring elements of a half-edge: (a) incident vertices and (b) incident triangles.*

```
accessVertices(int halfEdgeIndex):
  v_1 = toVertex(halfEdgeIndex)
  v_2 = toVertex(oppositeHalfEdge(halfEdgeIndex))
```

Triangles incident to an edge

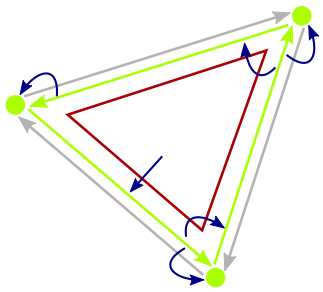
Incident triangles can be accessed in a similar way as for vertices; visible in figure 4.5b:

```
accessIncidentTriangles(int halfEdgeIndex):
  t_1 = adjacentTriangle(halfEdgeIndex)
  t_2 = adjacentTriangle(oppositeHalfEdge(halfEdgeIndex))
```

Vertices of a triangle

In order to have access to all vertices of a triangle in anticlockwise direction each half-edge uses the reference to the next half-edge and then the reference pointing to the vertex (visible in figure 4.6):

```
accessVerticesOfTriangle(int triangleIndex):
  int halfEdgeIndex = outgoingHalfEdge(triangleIndex)
  for (int i = 0; i < 3; ++i): # a triangle has three half-edges
    int vertexIndex = toVertex(halfEdgeIndex)
    # do something with vertexIndex
    halfEdgeIndex = nextHalfEdge(halfEdgeIndex)
```



(a)

Figure 4.6.: *Access to vertices of a triangle in anticlockwise direction.*

Local Mesh Modification

Next, there will be a rough outline of the modifications that need to be performed on the half-edge data structure when using topological operations.

Edge-split: This operation adds elements (one vertex, six half-edges and two triangles), nothing is removed and references are changed.

- Create new vertex at the midpoint of the edge
- Change the connectivity from the half-edges of the edge to connect the left and new vertex
- Create two half-edges between the new vertex and the right vertex
- Create four half-edges pointing to/from the vertex on top and bottom (two if it is a boundary edge)
- Create two new triangles (one if it is a boundary edge)
- Update references of the new half-edges to the correct triangles and vice versa
- Possibly update references of old triangles to correct half-edges
- Update references to the next half-edges for most of the half-edges

Edge collapse: This operation removes elements (one vertex, six half-edges and two triangles), nothing is added and references and one vertex position are changed.

- Set the position of the left vertex to the position of the new vertex (midpoint of the edge)

- Change the vertex of the half-edges referencing the right vertex to the new vertex
- Update half-edges connected from the top/bottom vertex to the left/right vertex
- Delete unnecessary elements: vertex, two triangles and six half-edges

Edge-flip: This operation does not remove or add elements; only references are changed.

The following elements have to be updated:

- For (right and left) vertices possibly outgoing half-edges
- For half-edges of the edge the vertex it points to and the reference to the next half-edge
- For triangles possibly the incident half-edge
- For outer adjacent half-edges the half-edge they point to and the reference to the incident triangle (just for top right and bottom left half-edges)

Move Vertex: For vertices a new position is assigned.

4.4. Data Preparation

The following preparation steps have to be used in order to create a valid triangle mesh and represent it in a way that it can be used by the application.

Triangulation

A common way to represent surfaces are triangle meshes. The meshes available to be imported into this application are not always triangulated as the supported file formats allow arbitrary polygons. Quite often quads are contained in meshes, too. To allow other polygons than just triangles in imported files, quadrangles are triangulated first. To keep this step simple files with other faces than triangles and quadrangles (i.e. polygons with more than 4 vertices) can not be processed.

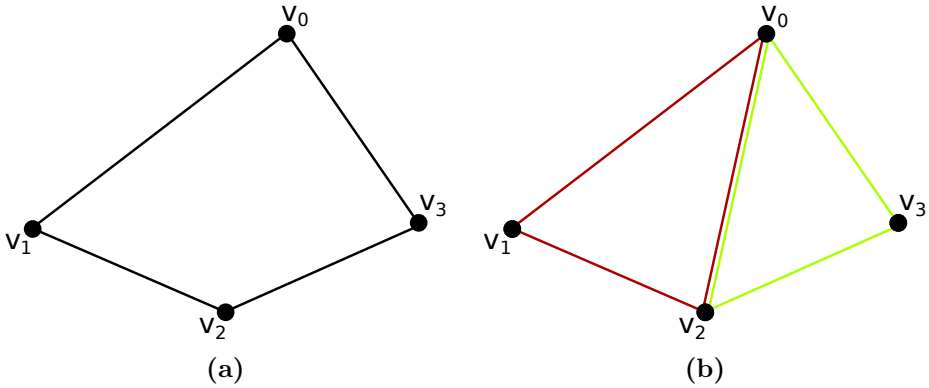


Figure 4.7.: *Triangulation of a quadrangle with $n = 4$ vertices. (a) Before triangulation and (b) after triangulation.*

The triangulation process can be seen in figure 4.7 for a quadrangle with $n = 4$ vertices. This quadrangle is divided into 2 triangles: $t(\mathbf{v}_0, \mathbf{v}_i, \mathbf{v}_{i+1})$, $i = 1, 2$.

Clean Data

The mesh data has to be cleaned in order to prevent unexpected behaviour by the application.

A first step is to delete duplicated vertices as otherwise some connectivity would not be recognisable. All vertices with the same position should exist only once and all faces referencing such a position reference the same vertex.

Next, triangles that have no surface area have to be removed from the mesh. Such triangles have for example the same vertex positions at least twice or all vertex position lie on the same line. Such faces have no specific direction in order to calculate the normals and no influence on the surface as their area is zero.

And finally, unused vertices – vertices that are never referenced – can be deleted. This is not necessary as they would be ignored for mesh operations as they have no neighbouring elements. In order to save memory and omit iterations over irrelevant elements they are removed.

Scaling and Shifting

Most meshes have different dimensions and their centre might be at different positions in the coordinate system. For visualisation and calculations the vertices of the meshes are moved to be centred around the origin of the coordinate system $(0, 0)$ and the mesh is scaled to fit into a cube around the origin with side lengths of two: $x, y, z \in [-1, 1]$.

These coordinate changes to the vertices ensure that each mesh is visible in the viewing area of the application. Additionally, the usage of the target edge lengths and approximation tolerance can be in relation to the cube with $x, y, z \in [-1, 1]$ and hence create similar results for meshes even if the scaling is different.

Create Data Structure

The half-edge data structure is created from a list of vertex positions and a list of triangles with indices to vertices. These lists are already prepared and valid for further processing.

First a list of *Vertex* elements is created containing the position but no half-edge reference.

Then, for each triangle the following steps are performed: For each edge of the triangle two *HalfEdge* elements are created for the list of half-edges if they do not exist yet. For each edge an entry in a map with vertex positions of the edge and the index is stored for testing whether a specific edge already exists and for accessing it. Depending on whether the half-edges for an edge already exist references for the half-edges to the incident vertices have to be set and also references to the half-edges for the vertices. Next, a *Triangle* with reference to one of these three half-edges is created. For all half-edges of the triangle the reference to the next half-edge has to be set in anticlockwise direction in the triangle as well as a reference to the newly-created triangle.

Calculate Normals

Each triangle and vertex of the mesh needs a normal vector. The calculation is performed according to chapter 3.9 for each triangle and vertex.

Create The Reference Mesh

Additionally, a reference mesh of the input mesh has to be stored. This structure is used later for back-projection of new vertex positions onto the original mesh. Simple lists of vertex positions and vertex indices for triangles are used.

4.5. Implementation Details

The tool is implemented in C++ using OpenGL and the NGL library for the visualisation of the surface mesh, Qt for the user interface and Boost Spirit.Qi for parsing. Doxygen is used for documentation.

The next section contains details about the implementation including classes used, their responsibilities and a class diagram.

Design of the Application

The structure of the implementation can be divided into several parts: One part is responsible for the visualisation, another for importing and exporting data, the next one for the user interface and the most important one for the representation of the mesh and remeshing.

The struct *Settings* contains all settings for the remeshing and the visualisation that can be defined in the user interface. *MainWindow* is responsible for the management of the application. The remeshing process is triggered here. It manages the communication between the user interface and other classes. In *UserInterface* the user interface with dock widgets and a menu for the application is being created and values are initialised from the struct *Settings*.

GLWidget is responsible for all functionalities concerning the visualisation. *MeshVisualization* creates and handles the vertex array objects for the mesh. It takes the mesh to traverse all its triangles and feature edges. Different layers are drawn for the final visualisation: the triangles are drawn both filled and with lines; feature edges are drawn separately. An offset is used for lines and triangles to improve their visibility when they have the same coordinates. Colour for curvature is defined for vertices rather than triangles and interpolated within the triangle areas. *CoordinateAxes* is used to show basic coordinate axes for orientation.

The class *Mesh* contains lists of vertices, half-edges and triangles; each of these elements contain indices of the other elements according to the half-edge data structure as described in section 4.3. These elements are of type *Vertex*, *HalfEdge* and *TriangleFace*. Additionally, functions are provided for accessing these elements and other elements in their neighbourhood in the mesh in order to modify them.

Remesher handles the complete remeshing process from curvature calculation over feature detection to modifying the mesh. Local mesh operations are performed using the functions available in the classes *MeshOperations* and *TangentialRelaxation* which are responsible for the operations edge-split, edge-collapse and edge-flip as well as the tangential relaxation. The back-projection onto the surface is performed using the class *BackProjection*. The classes *CurvatureCalculator* and *FeatureCalculator* calculate the curvature and detect feature edges for the mesh respectively; this information is used for remeshing.

The name space *HelperFunctions* contains additional functions for vector calculation (calculate surface area, angle, barycentre or normalised normal for a triangle) used by the classes dealing with the meshes.

The classes *JSONImporter* and *JSONExporter* are responsible for importing and exporting settings that can be defined in the user interface of the application. It includes the specification of a file path in order to automatically load a mesh when reading such a settings file.

PNGExporter handles exporting of the image of the mesh visible in the viewing area as PNG file.

Mesh importer classes (*OFFImporter*, *OBJImporter* and *PLYImporter*) are responsible for loading a mesh from a file using the boost Spirit.Qi library for parsing. It is verified that the available data is valid for a mesh and faces are processed in order to obtain a triangle mesh without duplicated or irrelevant data. As these classes have the same purpose they inherit from the abstract class *MeshImporter*. This class provides general functionality to prepare the data from the files and transforms vertex and face data to a triangle mesh represented by the half-edge data structure.

The exporter classes (*OFFExporter*, *OBJExporter* and *PLYExporter*) export the (remeshed) mesh to a file by traversing all vertices and triangles in the data structure of the mesh. These classes inherit from *MeshExporter*.

In figure 4.8 a simplified class diagram of the implemented application is shown to see the basic structure. It contains the previously described classes and their connections.

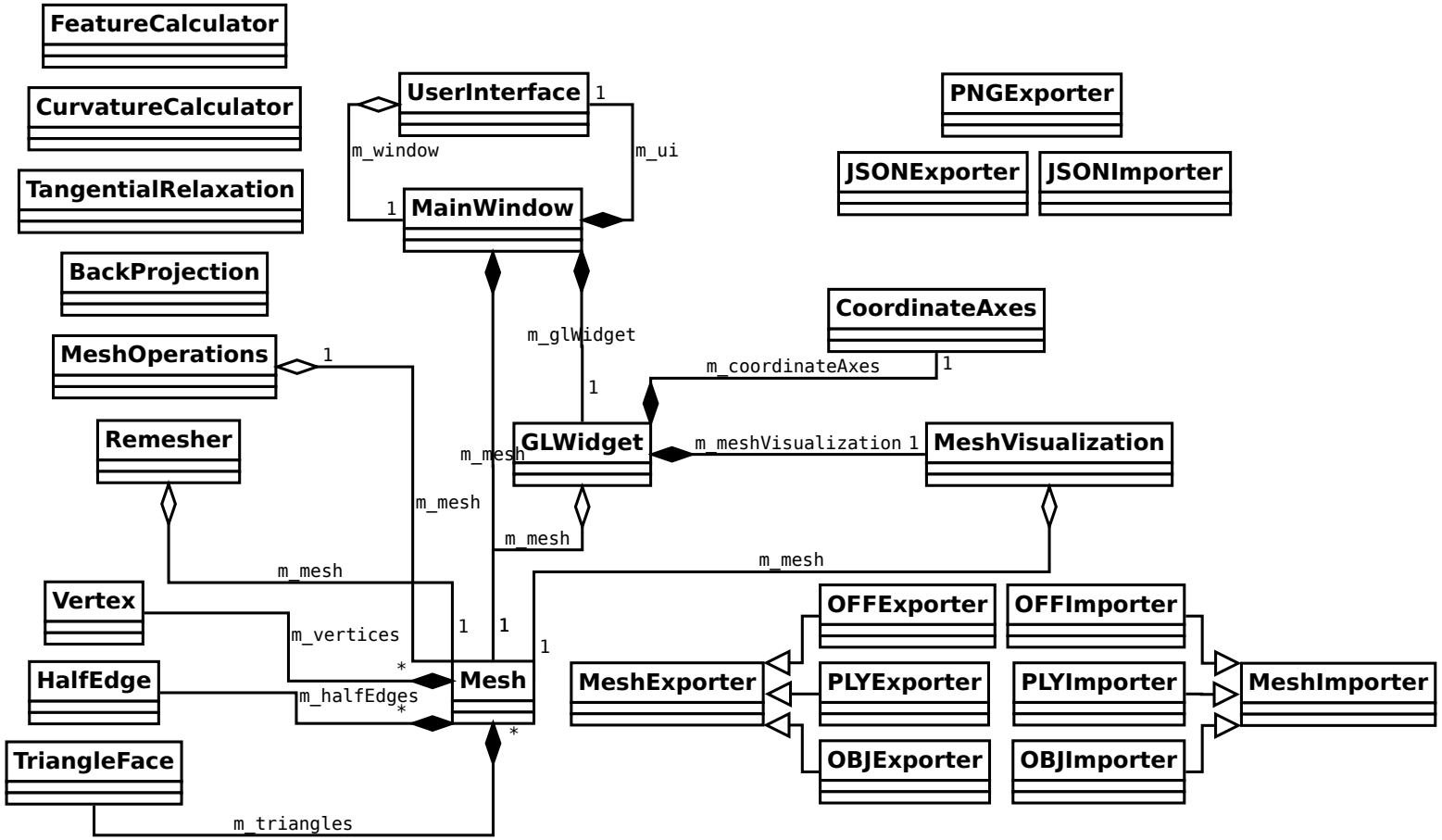


Figure 4.8.: *Class diagram for the remeshing application.*

4.6. User Interface

The user interface for the implemented application showing an example mesh can be seen in figure 4.9.

Interaction Techniques

In the application a manifold mesh with boundary can be loaded and remeshed. In addition, it is possible to specify values for the remeshing process. The remeshing type can be selected: uniform or curvature-adaptive. According to the chosen method a target edge length or an approximation tolerance can be set as well as the number of iterations, the minimum angle for feature detection, a gamma value for curvature calculation and others. The reference mesh can be reloaded to restart the remeshing process with different values.

Furthermore, it is possible to navigate in the viewing area by zooming, panning and rotating. Elements as contours can be hidden and colours for different areas/elements can be changed and curvature and feature edges can be visualised. In the viewing area the current number of vertices, edges and triangles is displayed.

Export and Import

Supported file formats for importing and exporting are OFF, PLY and OBJ (details about these formats in section 4.2). The application also provides the possibility to export the currently visible scene as a PNG image with metadata containing the current settings. JSON import and export is available for all settings in the user interface to restore a previous session. An example of such a file can be seen in appendix D.

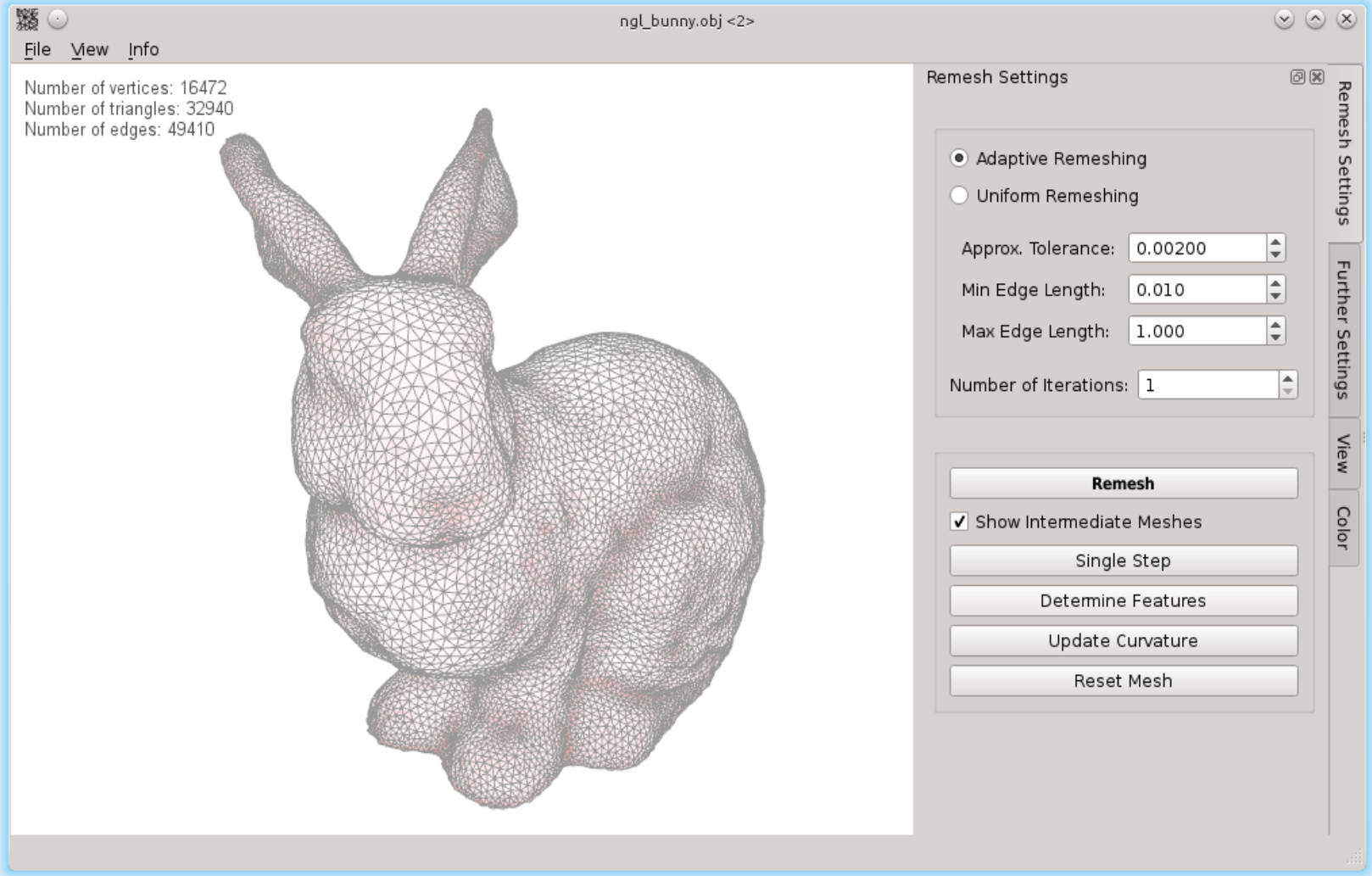


Figure 4.9.: *User interface with a processed example mesh.*

5. Remeshing Results

This chapter presents results obtained with the remeshing application for some simple geometry and a selection of common models used in computer graphics.

All images presented below were created with the application developed in this project. For the meshes images are presented of the original (triangulated) models, some remeshed versions of the models with different settings and visualisations created using values from feature detection and curvature approximation.

Usually fewer than five iterations for remeshing were used. Quite often, after one or two iterations a satisfactory result was achieved. Usually, back-projection was not used as the results for the models used in this project were satisfactory. However, using back-projection as a final step after many iterations might create a better result for some meshes as all vertices are moved back onto the original surface. Otherwise, smoothing created by the relaxation step might then not affect the general shape of the mesh that much anymore. Curvature is calculated only once before remeshing; curvature values for new created vertices are interpolated from their neighbouring vertices. Feature detection is not used except when mentioned. Sometimes it is difficult to choose appropriate values for parameters of adaptive remeshing to achieve an expected result. The quality of the created meshes depend on these settings.

The time required for the creation of the final meshes varies as the size of the original meshes is quite different. Small meshes (the meshes in the first section) need several milliseconds for their creation while larger meshes (for instance the Stanford Dragon) require several seconds. The number of elements (vertices, edges and triangles) can be seen for meshes in the available images.

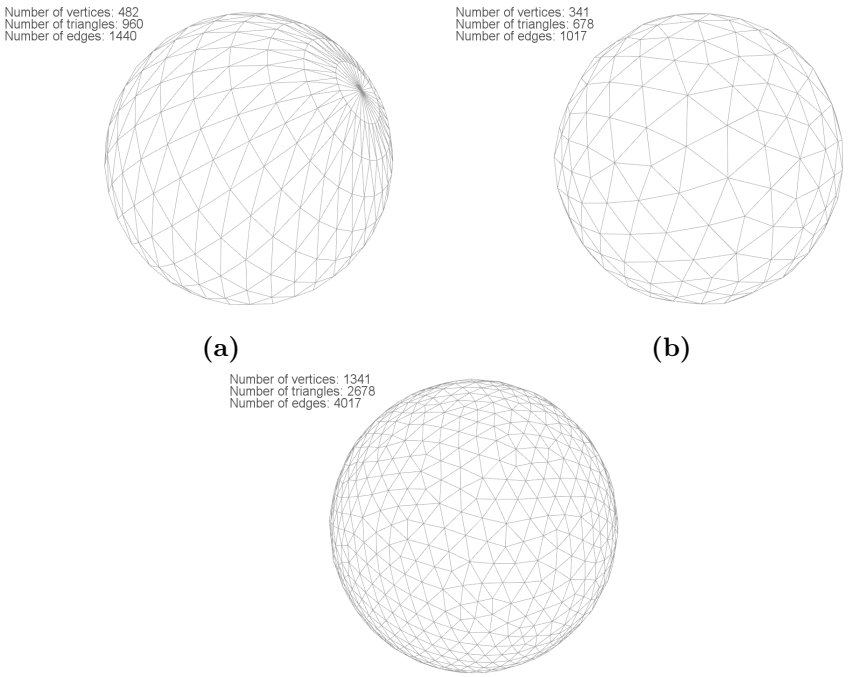


Figure 5.1.: *Sphere: (a) Original mesh, (b) and (c) uniform remeshing with different target edge lengths* (0.2 *and* 0.1).

Simple Geometry

This section presents some results for simple geometry to understand the behaviour of the process and to show results that can be expected.

Uniform Remeshing

As a sphere has no difference in curvature it is appropriate to perform uniform remeshing. Additionally, a sphere has neither features nor boundaries. Figure 5.1 shows a simple sphere before and after uniform remeshing for different target edge lengths. The result of remeshing is that the shape and size of the triangles are more regular. The valences of vertices are more regular (especially for the vertices a the very top and bottom of the original mesh).

Feature Preservation

A cube is an appropriate example to show feature detection and preservation. In figure 5.2 a cube is visible and detected feature lines are visualised in blue. Uniform remeshing with different target edge lengths is performed on this cube. The features are not destroyed and the general shape of the cube is preserved.

Adaptive Remeshing

For curvature-adaptive remeshing the curvature of meshes is to be estimated for the final edge lengths. Below, examples for a stretched sphere (figure 5.3) and a torus (figure 5.4) show results for both uniform and adaptive remeshing. Low curvature is visualised white, high curvature red. It is visible that red areas have a higher vertex sampling than brighter areas when using adaptive remeshing.

Boundary Handling

Boundaries can be handled in two different ways: the general shape can be preserved or tangential relaxation can lead to a smooth mesh. Figure 5.5 shows a sphere with a hole. Different images show that boundaries can be detected as features and that boundaries have no valid curvature (visualised in turquoise instead of red). When performing resmeshing the boundary can be preserved as visible in figure 5.5d or tangential relaxation can be used as for figure 5.5e.

Stanford Bunny

The Stanford Bunny is a model often used in computer graphics. Figure 5.6a shows the original mesh used for remeshing. It consists of 34835 vertices, 104499 edges and 69666 triangles. The remaining images in figure 5.6 show the bunny with curvature calculation and uniform or adaptive remeshing.

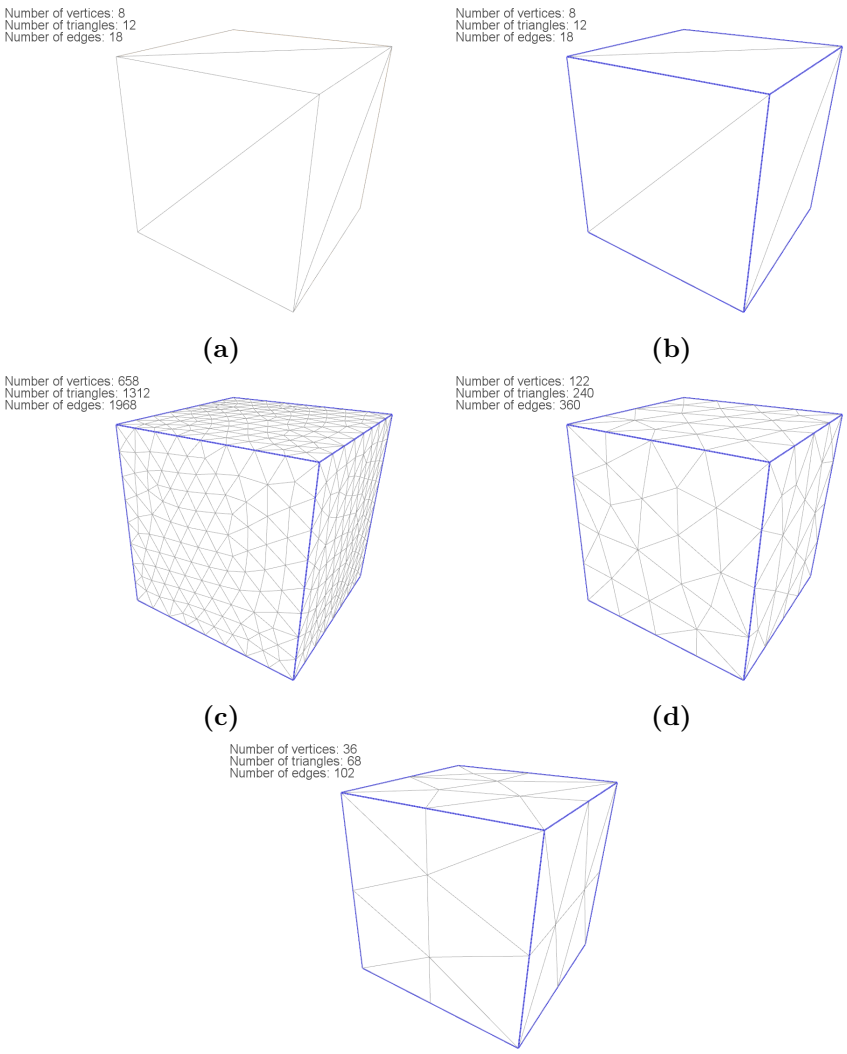


Figure 5.2.: *Cube: (a) Original mesh, (b) with feature detection, (c) - (e) uniform remeshing with different target edge lengths* (0.2, 0.5 *and* 1.0).

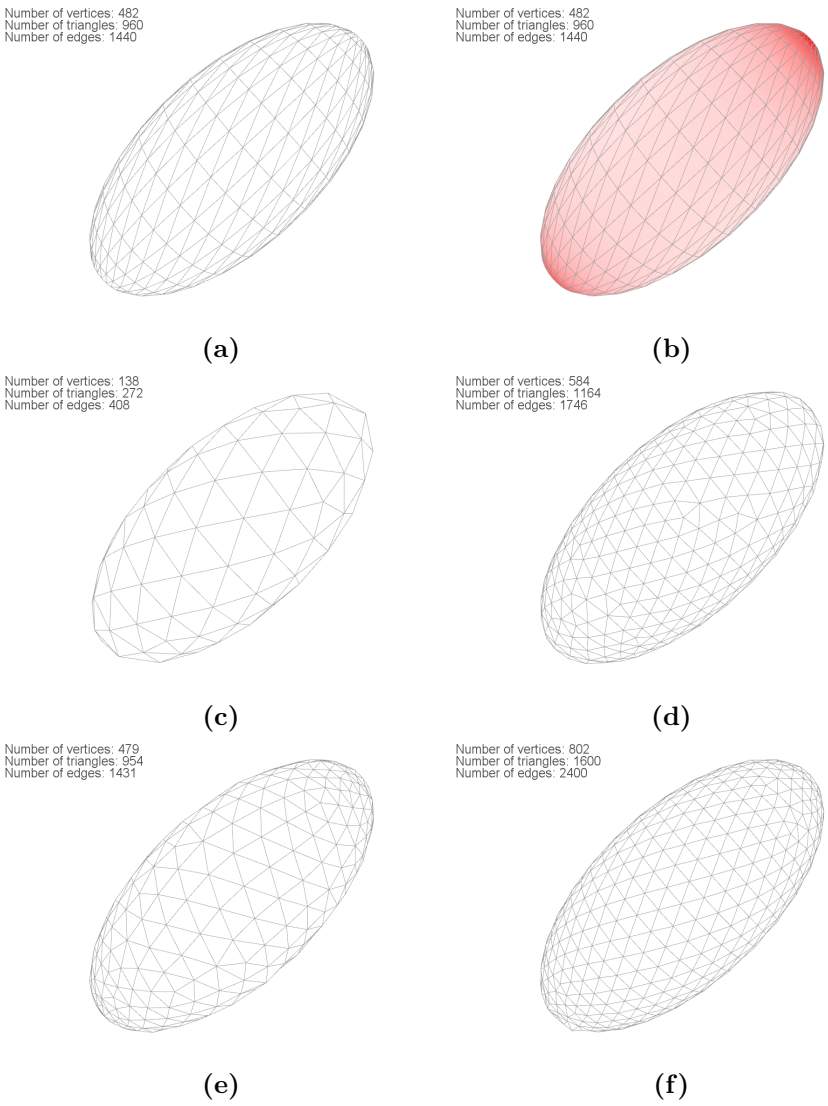


Figure 5.3.: *Stretched sphere: (a) Original mesh, (b) with curvature calculation, (c) and (d) uniform remeshing with target edge lengths* 0.2 *and* 0.1*, (e) and (f) curvature-adaptive remeshing with* $\epsilon = 0.002$*,* $\gamma = 2$ *and* $\epsilon = 0.002$*,* $\gamma = 1$*.*

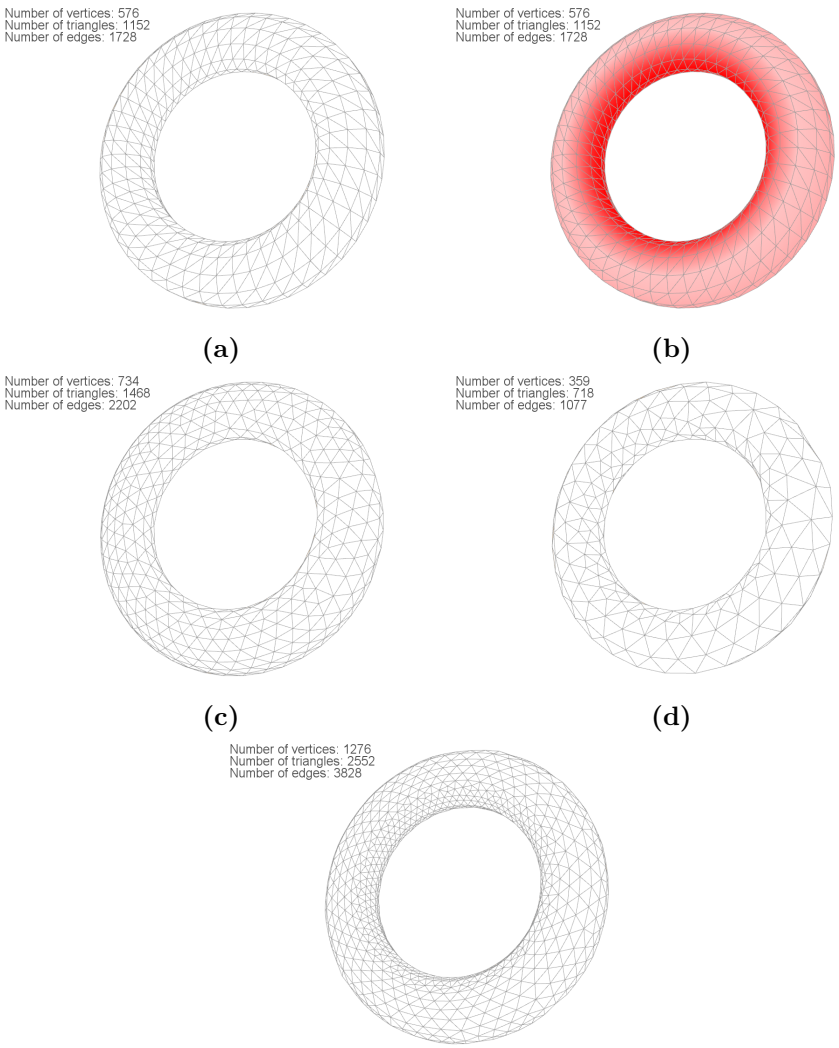


Figure 5.4.: *Torus: (a) Original mesh, (b) with curvature detection, (c) after uniform remeshing with target edge length* 0.1*, (d) and (e) after adaptive remeshing with* $\epsilon = 0.008$*,* $\gamma = 3$ *and* $\epsilon = 0.002$*,* $\gamma = 3$*.*

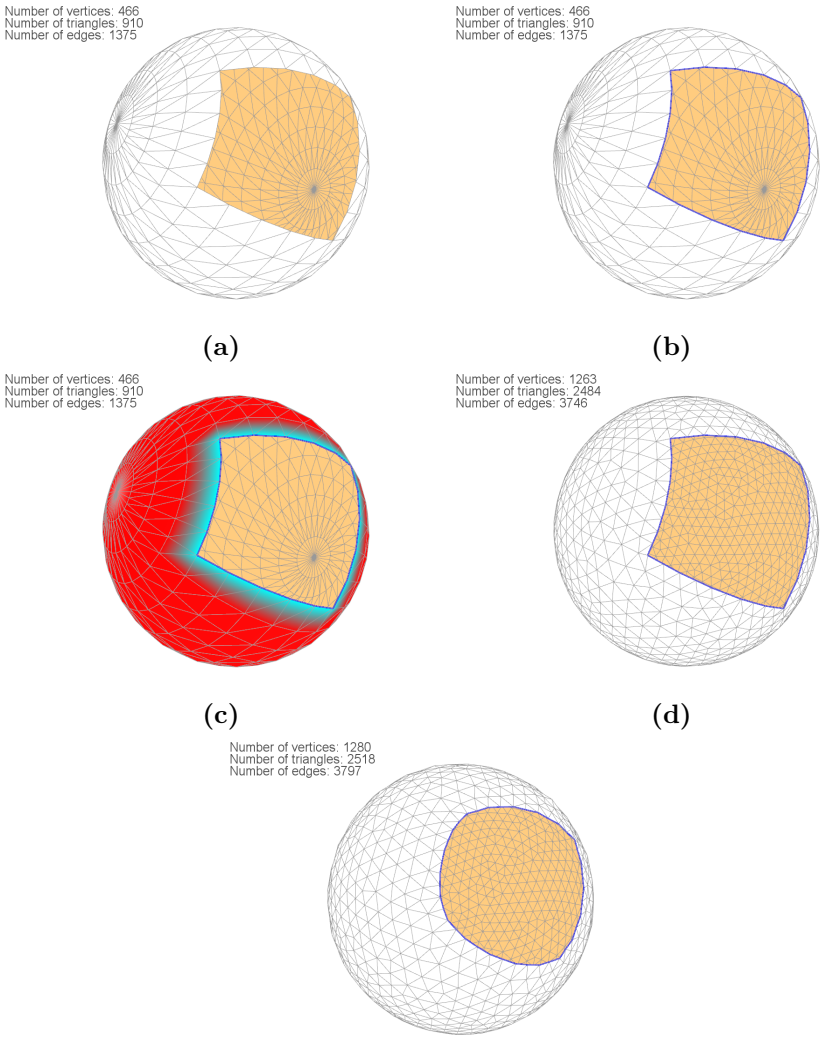


Figure 5.5.: *Sphere with hole: (a) Original mesh, (b) with feature detection, (c) with curvature calculation, (d) uniform remeshing with feature/boundary preservation and (e) with smoothing.*

All remeshing results offer a quite good approximation of the surface while using fewer elements than the original mesh has. Results for uniform and adaptive remeshing with similar amount of elements differ as more elements are used in areas of high curvature and fewer in areas of low curvature for adaptive remeshing. Thus, with adaptive remeshing a better approximation for a surface is created.

Stanford Dragon

The original mesh used for the Stanford Dragon consists of 50000 vertices, 150000 edges and 100000 triangles. Figure 5.7 shows remeshing results for uniform and adaptive remeshing.

For this model it was noticed that areas with very small elements exist. These need many iterations to be removed from the mesh. In figure 5.7e the result for uniform remeshing with 100 iterations is visible. Even here, there are areas on the mesh with small triangles. Another observation was that the adaptive remeshing was quite sensitive to noise. Especially figure 5.7g shows accumulation of triangles in an area where a high curvature value was detected. Smoothing of curvature values of vertices with considering their neighbourhood might avoid such effects.

Fandisk Model

The fandisk model is a good model to demonstrate feature detection and preservation as sharp edges are present. Figure 5.8 shows the results obtained for this model. The original model has 6475 vertices, 19419 edges and 12946 triangles. For curvature calculation two different results are visible using different parameter values. The results obtained by uniform remeshing is a mesh with a strongly decimated number of elements; it is visible that it is important to consider the features as otherwise sharp edges and corners would be destroyed. The result for adaptive remeshing shows that more triangles are used in areas of higher curvature. Remeshing for this model took less than one second on a several-year-old consumer-grade laptop.

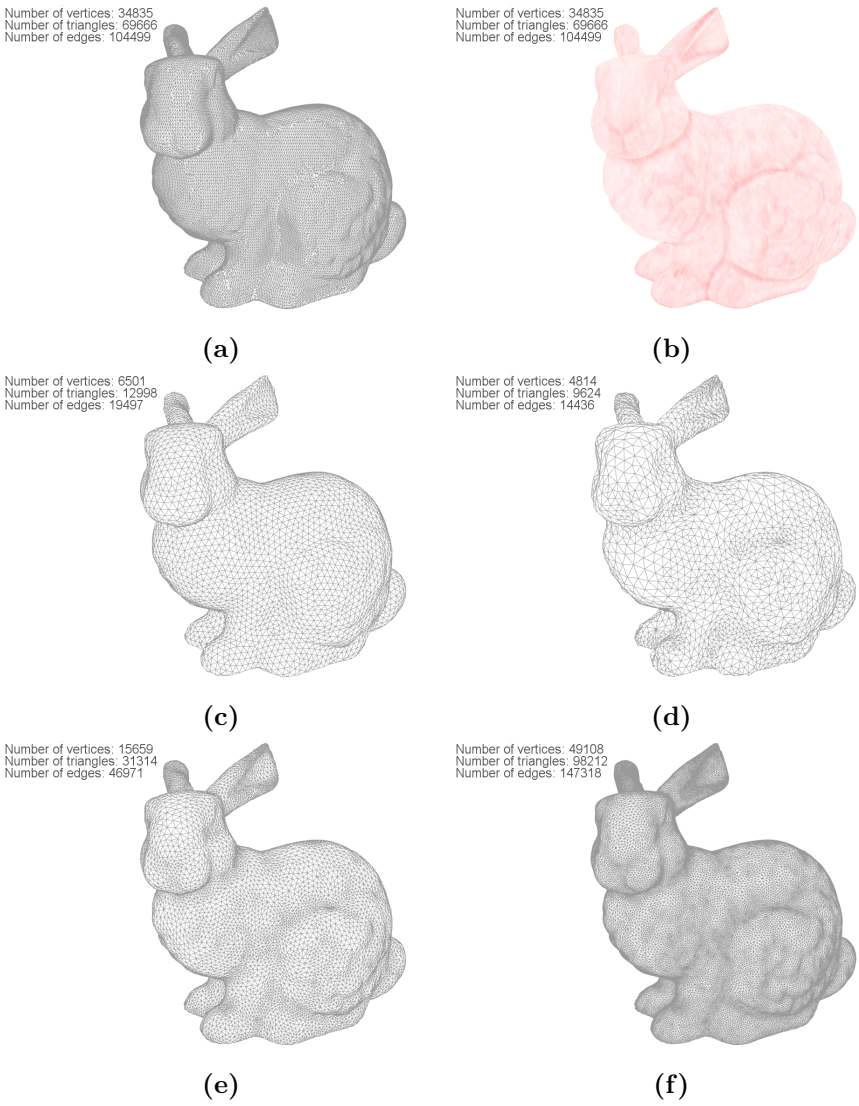


Figure 5.6.: *Stanford Bunny: (a) Original mesh, (b) with curvature calculation, (c) uniform remeshing with target edge length* 0.04 *and (d) - (f) adaptive remeshing for different parameters: (d)* $\epsilon = 0.01$, $\gamma = 1$*; (e)* $\epsilon = 0.002$, $\gamma = 1$*; (f)* $\epsilon = 0.0005$, $\gamma = 1$.

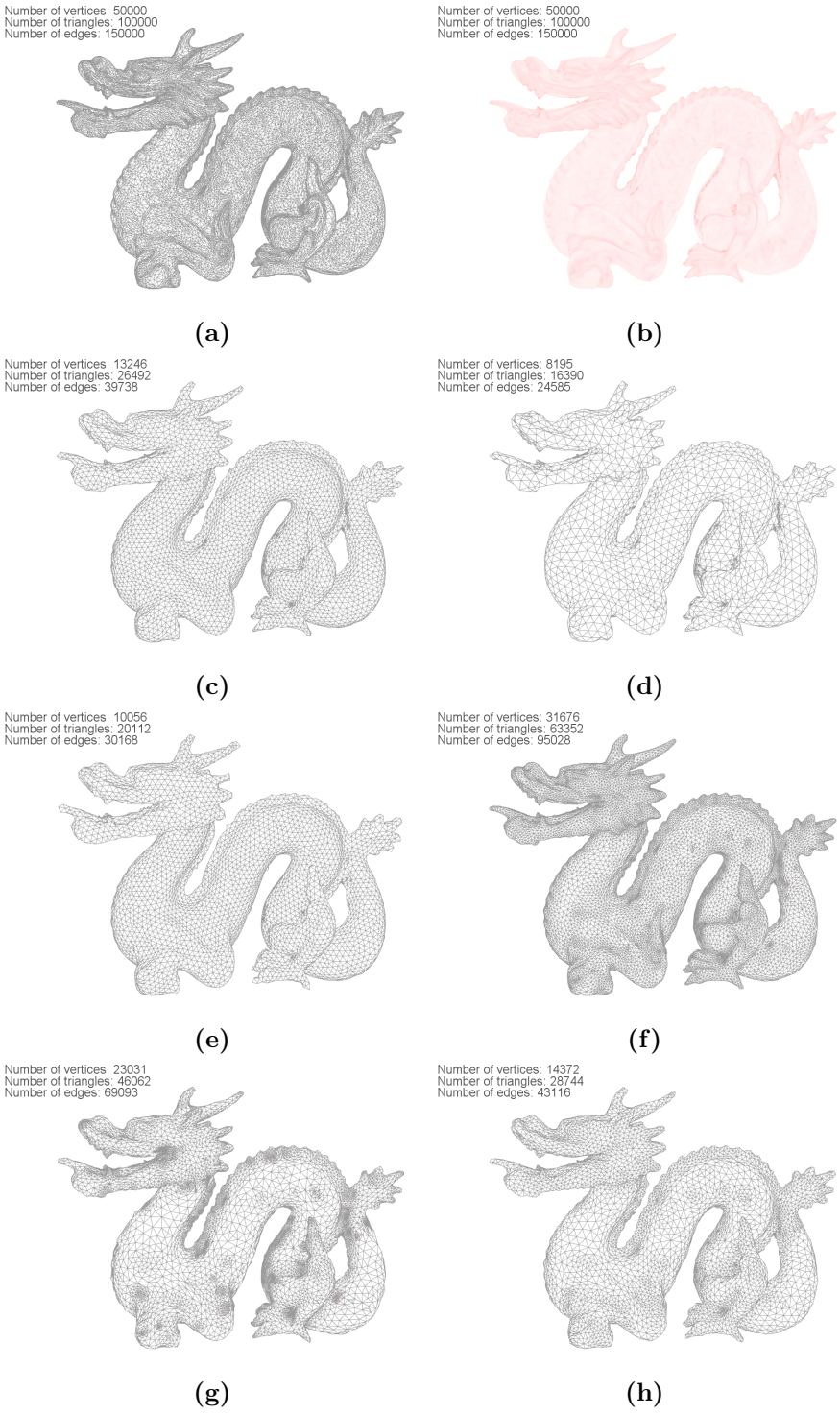


Figure 5.7.: *Stanford Dragon: (a) Original mesh, (b) with curvature calculation, (c) and (d) uniform remeshing with target edge length* 0.03 *and* 0.05*, (e) uniform remeshing for* 100 *iterations and (f) - (h) adaptive remeshing for different parameters: (f)* $\epsilon = 0.002$*,* $\gamma = 1$*; (g)* $\epsilon = 0.01$*,* $\gamma = 1$*; (h)* $\epsilon = 0.008$*,* $\gamma = 1$ *and minimum edge length* 0.02*.*

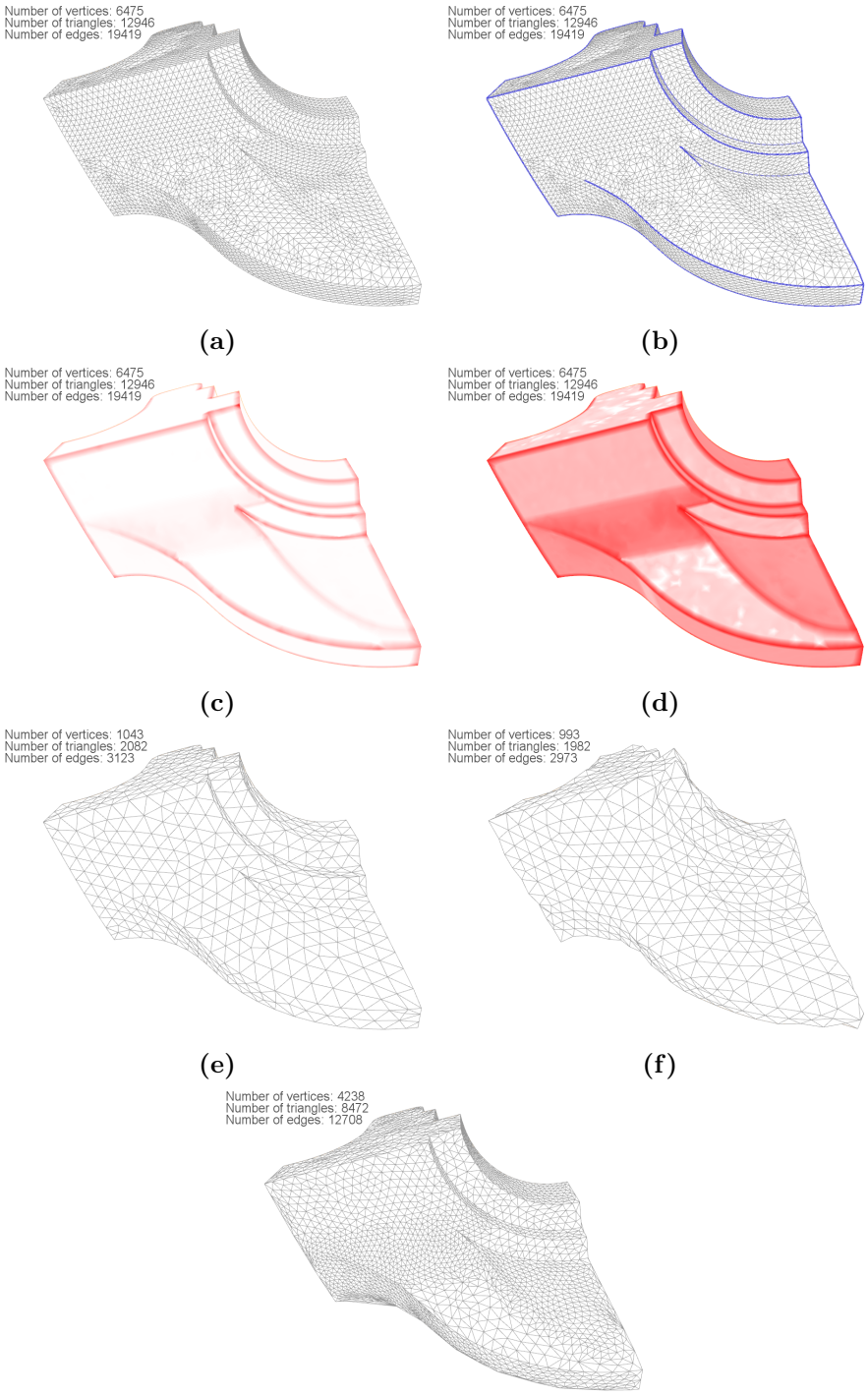


Figure 5.8.: *Fandisk Model: (a) Original mesh, (b) feature extraction, (c) and (d) with curvature calculation* ($\epsilon = 0.002$, $\gamma = 1$ *and* $\epsilon = 0.01$, $\gamma = 0.2$)*, (e) and (f) uniform remeshing with target edge length* 0.1 *with and without feature preservation and (g) adaptive remeshing with* $\epsilon = 0.01$, $\gamma = 0.2$.

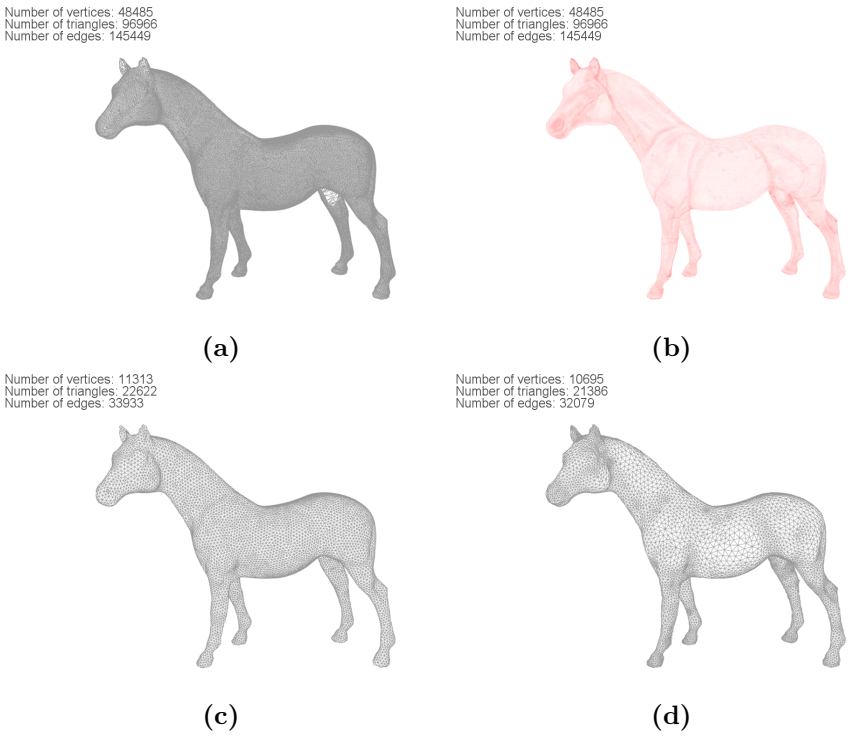


Figure 5.9.: *Horse: (a) Original mesh, (b) with curvature calculation, (c) uniform remeshing with target edge length* 0.02*, (d) adaptive remeshing with* $\epsilon = 0.002$*,* $\gamma = 1$*.*

Horse

The horse model is oversampled and consists of 48485 vertices, 145449 edges and 96966 triangles. Figure 5.9 shows the results for this mesh using about a fifth of the elements for the resulting meshes. For uniform and adaptive remeshing about the same number of elements is used. Especially small details (ears and legs) have a finer approximation with adaptive remeshing.

Hammerhead

The original model for the hammerhead contained 2560 vertices, 7674 edges and 5116 triangles. For this model feature preservation helped to retain the general shape. An improvement is achieved by the tangential relaxation equalising the vertex distribution and creating more equal triangle shapes. As visible in figure 5.10 uniform remeshing is not able to preserve small details. The result obtained by adaptive remeshing, especially when more elements are used as for the original mesh, creates an additional smoothing of the surface.

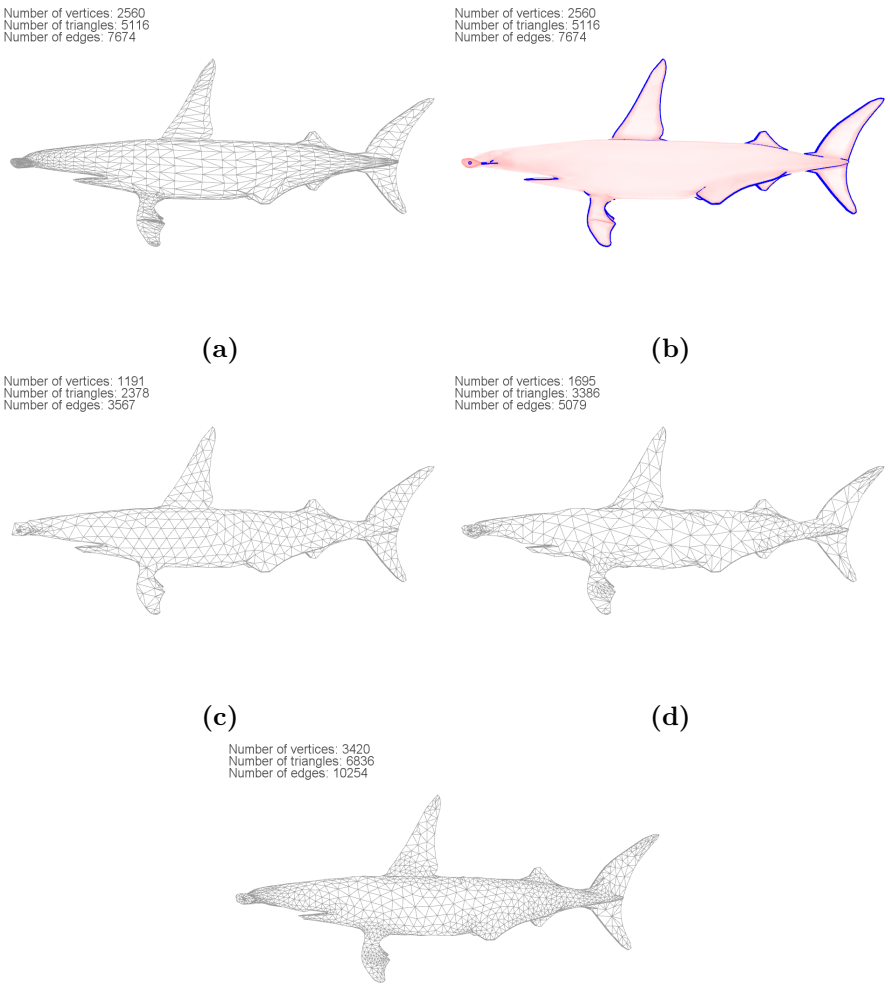


Figure 5.10.: *Hammerhead: (a) Original mesh, (b) with feature detection and curvature calculation, (c) uniform remeshing with target edge length* 0.05*, (d) and (e) adaptive remeshing with* $\epsilon = 0.01$*,* $\gamma = 1$ *and* $\epsilon = 0.002$*,* $\gamma = 1$*.*

6. Conclusion

6.1. Summary

The goal of this project was the creation of an interactive application for curvature-adaptive isotropic remeshing of given (triangulated) surface meshes.

This report provided an overview of some research in the area of remeshing. Based on the approach described in Dunyach *et al.* (2013) important background information was provided that is relevant for the general remeshing process and the implementation. The application was presented and details about the data structure used and important preprocessing steps were given. Finally, results obtained with the application were presented and analysed for both curvature-adaptive and uniform remeshing.

The implemented approach is based on local mesh modifications (edge-split, edge-collapse, edge-flip and vertex move) to generate a better mesh for a given input mesh. Both feature preservation and an approximated curvature for the underlying surface are considered for the final output. Meshes that can be handled by the application are not limited to closed 2-manifold triangle meshes that have genus zero; meshes of arbitrary genus with boundary containing triangles and quads can be imported. Different file formats (OFF, PLY and OBJ) can be used for import and export. Internally, meshes are represented by the half-edge data structure. The results that can be generated by this approach are of good quality, regular and curvature-adaptive.

6.2. Future Work

While the implemented remeshing application offers good results for remeshing of irregular meshes further improvements for both the remeshing process and the application are still possible. The main interests for improvements of the application would be related to the speed of the remeshing process and the interaction possibilities for users. There is also still potential to enhance the quality of the remeshing result.

The results obtained during this project can not be achieved in the same time as mentioned in the approach described in Dunyach *et al.* (2013).

To improve the speed of the algorithm it would be advisable to use an external library (such as OpenMesh or CGAL) for the half-edge data structure and the local mesh operations. The application would be computationally more effective and the memory consumption better as these libraries are specialised to these operations and use special techniques like generative programming concepts (Botsch *et al.* 2002).

The back-projection of vertices onto the original mesh is quite expensive in this implementation. In Botsch *et al.* (2010) it is stated that techniques using parameterisation, especially global parameterisation, compared to naive direct projection are expensive. However, in Surazhsky and Gotsman (2003) it is mentioned that the projection of vertices onto the original surface is quite expensive. This can be seen in the results for the implementation of this project. A faster method should be chosen.

Moreover, parallelisation could be used as it is done in the approach by Fuhrmann *et al.* (2010). With multi threading the performance could be increased.

Considering interaction possibilities the application is quite limited. A great enhancement would be if the application would allow interactive modification (deformation, modelling, sculpting) of the mesh and immediate remeshing.

Right now the whole mesh is getting remeshed. If selected regions only were able to be remeshed it would be possible to handle larger meshes, e.g. for deformation of the mesh in a specific part only local remeshing would be sufficient.

Features are extracted algorithmically; by user selection of edges and vertices specific areas could be preserved and prevent a destruction of features not recognised.

While the target edge length/approximation tolerance can be specified it might be more user friendly to specify the number of elements for the final mesh as in Fuhrmann *et al.* (2010) to receive a mesh with a specific resolution. As edge lengths and the approximation tolerance are relative to the mesh size it is difficult to choose an appropriate value. In the implementation this was handled by scaling all models to fit into a box of the same size.

According to Dunyach *et al.* (2013) the approximation tolerance is not satisfied exactly due to the sizing field calculation. A higher quality mesh could be achieved with a more advanced vertex shift.

For both curvature calculation and feature detection quite simple methods were chosen. Using a more complex curvature approximation with smoothing and a different feature extraction approach which are both less sensitive to noise a better adaptive sampling might be created as well as a better shape preservation. This would have influence on the performance, though.

Many available meshes are supported by the application. An extension would be the support for import of meshes with arbitrary faces (as arbitrary polygons can be defined in the input files) and support for non-manifold meshes. Arbitrary faces could be handled simply by using an extended triangulation approach; the support for non-manifold meshes would require a different data structure. This however, might have an influence on the performance of the algorithms.

At the moment vertex and face normals are calculated by the application. A better way to deal with these normals would be if they were taken from the input file and interpolated while modifying the mesh. Especially corner normals would be handled in a better way.

Bibliography

Alliez P., de Verdire E., Devillers O. and Isenburg M., May 2003. Isotropic Surface Remeshing. In *Shape Modeling International, 2003*, 49–58.

Alliez P., Meyer M. and Desbrun M., 2002. Interactive Geometry Remeshing. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, New York, NY, USA. ACM, 347–354.

Alliez P., Ucelli G., Gotsman C. and Attene M. 2008. 53–82. Recent Advances in Remeshing of Surfaces. In *Shape analysis and structuring*, Springer.

Attene M., Falcidieno B., Rossignac J. R. and Spagnuolo M., 2003. Edge-Sharpener: Recovering Sharp Features in Triangulations of Non-Adaptively Re-Meshed Surfaces.

Bommes D., Bruno L. and others , 2012. State of the Art in Quad Meshing.

Botsch M. and Kobbelt L., 2004. A Remeshing Approach to Multiresolution Modeling. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '04, New York, NY, USA. ACM, 185–192.

Botsch M., Kobbelt L., Pauly M., Alliez P. and Lévy B., 2010. *Polygon Mesh Processing*. CRC press.

Botsch M., Steinberg S., Bischoff S. and Kobbelt L., 2002. Openmesh - a Generic and Efficient Polygon Mesh Data Structure.

Chen L. and Holst M., 2011. Efficient Mesh Optimization Schemes Based on Optimal Delaunay Triangulations. *Computer Methods in Applied Mechanics and Engineering*, **200**(9), 967–984.

Dey T. K., Edelsbrunner H., Guha S. and Nekhayev D. V., 1999. Topology Preserving Edge Contraction. *Publ. Inst. Math.(Beograd)(NS)*, **66**(80), 23–45.

Dunyach M., Vanderhaeghe D., Barthe L. and Botsch M., 2013. Adaptive Remeshing for Real-Time Mesh Deformation. *Eurographics Short Papers*, 29–32.

Dyn N., Hormann K., Kim S.-J. and Levin D. 2001. Optimizing 3D Triangulations Using Discrete Curvature Analysis, 135–146. Mathematical Methods for Curves and Surfaces. Vanderbilt University.

Eberly D., 1999. Distance Between Point and Triangle in 3D. *http://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf*.

Frey P. J., 2000. About Surface Remeshing.

Fu Y. and Zhou B., August 2009. Direct Sampling on Surfaces for High Quality Remeshing. *Comput. Aided Geom. Des.*, **26**(6), 711–723.

Fuhrmann S., Ackermann J., Kalbe T. and Goesele M., 2010. Direct Resampling for Isotropic Surface Remeshing. In *VMV*. Citeseer, 9–16.

Hoppe H., DeRose T., Duchamp T., McDonald J. and Stuetzle W., 1993. Mesh Optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 19–26.

Kobbelt L. and Botsch M., 2003. Feature Sensitive Mesh Processing. In *Proceedings of the 19th Spring Conference on Computer Graphics*, SCCG '03, New York, NY, USA. ACM, 17–22.

Kobbelt L. P., Bareuther T. and Seidel H.-P., 2000. Multiresolution Shape Deformations for Meshes with Dynamic Vertex Connectivity. In *Computer Graphics Forum*, volume 19. Wiley Online Library, 249–260.

Lévy B., Petitjean S., Ray N. and Maillot J., 2002. Least Squares Conformal Maps for Automatic Texture Atlas Generation. In *ACM Transactions on Graphics (TOG)*, volume 21. ACM, 362–371.

Meyer M., Desbrun M., Schröder P. and Barr A. H. 2003. 35–57. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. In *Visualization and mathematics III*, Springer.

Petitjean S., 2002. A Survey of Methods for Recovering Quadrics in Triangle Meshes. *ACM Computing Surveys (CSUR)*, **34**(2), 211–262.

Surazhsky V., Alliez P. and Gotsman C., 2003. Isotropic Remeshing of Surfaces: a Local Parameterization Approach.

Surazhsky V. and Gotsman C., 2003. Explicit Surface Remeshing. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, SGP '03, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, 20–30.

Turk G., 1992. Re-Tiling Polygonal Surfaces. *ACM SIGGRAPH Computer Graphics*, **26**(2), 55–64.

Vivodtzev F., Bonneau G.-P. and Le Texier P., 2005. Topology-Preserving Simplification of 2D Nonmanifold Meshes with Embedded Structures. *The Visual Computer*, **21**(8-10), 679–688.

Vorsatz J., Rössl C. and Seidel H.-P., 2003. Dynamic Remeshing and Applications. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, New York, NY, USA. ACM, 167–175.

Vorsatz J., Rössl C., Kobbelt L. P. and Seidel H.-P., 2001. Feature Sensitive Remeshing. In *Computer Graphics Forum*, volume 20. Wiley Online Library, 393–401.

Zilske M., Lamecker H. and Zachow S., 2007. *Adaptive Remeshing of Non-Manifold Surfaces*. Konrad-Zuse-Zentrum für Informationstechnik.

A. OFF Example File

```
OFF
# This is a triangulated cube with vertex and face data

# number of vertices, faces and edges
# (value for edges is not used)
8 12 0

# vertices
0.0 1.0 0.0
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 1.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0

# faces
3  0 3 2
3  0 2 1
3  4 5 6
3  4 6 7
3  0 1 5
3  0 5 4
3  7 6 2
3  7 2 3
3  0 4 7
3  0 7 3
3  5 1 2
3  5 2 6
```

B. PLY Example File

```
ply
format ascii 1.0
comment This is a triangulated cube with vertex and face data
element vertex 8
property float x
property float y
property float z
element face 12
property list uchar int vertex_index
end_header

0.0 1.0 0.0
0.0 0.0 0.0
1.0 0.0 0.0
1.0 1.0 0.0
0.0 1.0 1.0
0.0 0.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0

3  0 3 2
3  0 2 1
3  4 5 6
3  4 6 7
3  0 1 5
3  0 5 4
3  7 6 2
3  7 2 3
3  0 4 7
3  0 7 3
3  5 1 2
3  5 2 6
```

C. OBJ Example File

```
# This is a triangulated cube with vertex and face data

# vertices
v  0.0 1.0 0.0
v  0.0 0.0 0.0
v  1.0 0.0 0.0
v  1.0 1.0 0.0
v  0.0 1.0 1.0
v  0.0 0.0 1.0
v  1.0 0.0 1.0
v  1.0 1.0 1.0

# faces
f  1 4 3
f  1 3 2
f  5 6 7
f  5 7 8
f  1 2 6
f  1 6 5
f  8 7 3
f  8 3 4
f  1 5 8
f  1 8 4
f  6 2 3
f  6 3 7
```

D. JSON File with Example Settings


```
{
    "Approximation Tolerance": 0.0020000000949949026,
    "Back Projection": false,
    "Background Color": [
        1,
        1,
        1,
        1
    ],
    "Boundaries as Features": true,
    "Curvature Color": [
        1,
        0,
        0,
        1
    ],
    "Draw Filled Triangles": true,
    "Draw Triangle Contours": true,
    "Feature Angle": 50,
    "Feature Color": [
        0,
        0,
        1,
        1
    ],
    "Gamma": 1,
    "Intermediate Meshes": true,
    "Loaded File": "bunny.obj",
    "Max Length": 1,
    "Min Length": 0.0099999997764825821,
    "Normal Length": 0.05000000074505806,
    "Number of Iterations": 1,
    "Random Order": false,
```

```
    "Recalculate Curvature": false,
    "Show CoordinateAxes": false,
    "Show Curvature": false,
    "Show Features": true,
    "Show Reference Mesh": false,
    "Show Remeshed Mesh": true,
    "Show Triangle Normals": false,
    "Show Vertex Normals": false,
    "Surface Back Color": [
        1,
        0.80000001192092896,
        0.5,
        1
    ],
    "Surface Front Color": [
        1,
        1,
        1,
        1
    ],
    "Tang. Relax. for Boundaries": false,
    "Tang. Relax. for Features": false,
    "Target Edge Length": 0.10000000149011612,
    "Triangle Contour Color": [
        0.60000002384185791,
        0.60000002384185791,
        0.60000002384185791,
        1
    ],
    "Triangle Normal Color": [
        0,
        0.80000001192092896,
        0.80000001192092896,
        1
    ],
    "Use Adaptive Remeshing": true,
    "Use Matte Shading": true,
    "Vertex Normal Color": [
        0.80000001192092896,
        0,
        0.80000001192092896,
        1
    ]
}
```