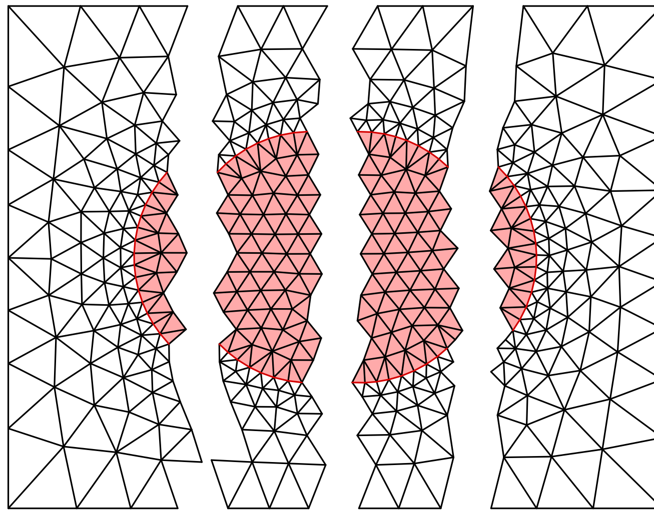
# Parallelization of Deformable Simplicial Complexes

Mark Viinblad Jensen
mark@markjensen.dk



Master of Science Thesis

Supervisor: Kenny Erleben
kenny@diku.dk

University of Copenhagen, Denmark
Department of Computer Science

4 December, 2013

**Abstract**

Simulations are used in many fields of science and they can be useful on the entire size scale, from elementary particles to planets of the universe. Deforming curves and surfaces occur in everyday situations, e.g. the surface of water, clothes, glass blowing and clay modelling. Deformable Simplicial Complex (DSC) is a method for the group of curves and surfaces called interfaces. A design for a parallel implementation of DSC is proposed. The implementation is highly configurable because quality measures, iteration stop criteria, interpolators etc. can be changed from the application level. Mesh operations are expressed using topology algebra. A default implementation of the DSC algorithm is provided, but an application specific algorithm can be implemented while preserving the parallel aspect. A slabbing method is implemented to create submeshes that can be worked on in parallel. The level of modularity means that even if some parts need to be replaced, other parts can be reused. Several demo applications have shown the flexibility of the implemented framework. A scaling demo gained a speedup of 0.96 for 1 process, 1.47 for 2 processes, 1.90 for 3 processes and 2.32 for 4 processes. This gives CPU utilizations of 0.96, 0.73, 0.63 and 0.58. It has been shown using another demo, which gains a speedup of 3.69 with 4 processes (CPU utilization of 0.92), that the bottleneck is when there are too few changes in the mesh per time step. The quality of the sequential algorithm and the parallel algorithm are comparable.

# Contents

**1 Introduction** **3**
1.1 Formalities . . . 3
1.2 Previous Work . . . 3
1.3 Overview of Thesis . . . 4
1.4 Notation Summary . . . 5

**2 Simulations** **6**
2.1 Simulation Framework . . . 6
2.2 Requirements . . . 7

**3 Simplicial Complexes** **10**
3.1 Simplex Relations and Operations . . . 11
3.2 Discrete Manifold . . . 14

**4 Mesh** **16**
4.1 Mesh Functions . . . 16
4.2 Mesh Operations . . . 16
4.3 Quality Measures . . . 20

**5 Deformable Simplicial Complex** **22**
5.1 Movement . . . 23
5.2 Coarsening . . . 23
5.3 Refinement . . . 23
5.4 Optimization . . . 24
5.5 Smoothing . . . 24
5.6 Algorithm . . . 24

**6 Mesh Decomposition** **27**
6.1 Red-Black Ordering . . . 27
6.2 Static and Dynamic Task Assignment . . . 28
6.3 Slabbing . . . 29
6.4 Locked Simplices . . . 31
6.5 Mesh Operation Restrictions . . . 34
6.6 Parallel Algorithm . . . 38

**7 Mesh Algorithms** **40**
7.1 Edge Flip . . . 40
7.2 Loop Subdivion . . . 40
7.3 Local Retrianguation . . . 42
7.4 Edge Split . . . 45
7.5 Edge Collapse . . . 47
7.6 Summary . . . 47

**8 Implementaton** **50**
8.1 Cross-Platform Support . . . 50
8.2 Simplex Programming Interface . . . 50
8.3 Attribute Vectors . . . 51
8.4 Mesh Data Structure . . . 51
8.5 Slabbing . . . 52

8.6 Interface and Boundaries . . . 52
8.7 Mesh Operations . . . 54
8.8 Mesh Quality . . . 54
8.9 Interpolators . . . 54
8.10 Sequential DSC and Parallel DSC . . . 54
8.11 Visualization . . . 54

**9 Experiments** **56**
9.1 Translation . . . 56
9.2 Scaling . . . 56
9.3 Rotation . . . 57
9.4 Vortex . . . 57
9.5 Interpolation . . . 57
9.6 Enright's Tests . . . 57
9.7 Speedup . . . 58
9.8 Quality . . . 59

**10 Results** **60**
10.1 Translation . . . 60
10.2 Scaling . . . 62
10.3 Rotation . . . 63
10.4 Vortex . . . 65
10.5 Interpolation . . . 66
10.6 Enright's Tests . . . 68
10.7 Speedup . . . 71
10.8 Quality . . . 74

**11 Discussion** **76**

**12 Conclusion** **79**

**13 Further Work** **80**

# 1 Introduction

## 1.1 Formalities

This document is the 30 ECTS points master's thesis project of the student Mark Viinblad Jensen. The project is conducted at the Computer Science department of Copenhagen University, starting May 2013 and ending December 2013. Kenny Erleben is the primary supervisor and Ulrik Bonde is the secondary supervisor. The author is a master student at the Department of Computer Science at Copenhagen University. The reader should have a similar or a related scientific background with an interest in simulations.

## 1.2 Previous Work

The study of simulations begins with a real-world problem which is converted to a mathematical model. This continuous mathematical model is discretized to create a discrete model that can be used on a computer. Some discrete models are better for some types of simulations than others. In this thesis, the focus is on unstructured triangulated meshes. These do often not allow triangles (or tetrahedra in three dimensions) to invert. Large deformations are therefore difficult to handle because they cause the deformation function to become ill-conditioned and the model becomes numerically unstable. In [7], the tetrahedral elements are allowed to invert, but this approach is not taken in this thesis. Significant effort has been placed into making the methods for non-inverting tetrahedra more robust using various techniques. In [2], an extension to Langrangian finite elements method is introduced to allow for large plastic deformations of solid objects. When the basis functions that governs the deformation becomes ill-conditioned, a remeshing subroutine remeshes the simulation domain to create new high-quality elements. One of the test examples takes more than one week to compute, where remeshing took 13% of the time. They explain that the long computation time is because they have not removed debugging code and not optimized the code, but they expect the method to be almost as fast as similar methods, although they do not provide any further evidence for this statement. In this thesis, however, the entire mesh is not retriangulated, but instead optimization operations will work on local subsets of the mesh and thus improving the quality of the entire mesh. In [4], a parallel algorithm for optimizing tetrahedral meshes is proposed. The method divides the mesh into submeshes and uses local coarsening and refinement operations to improve the mesh quality. On the boundaries, synchronization techniques are used to move tetrahedra from one submesh to its neighbour if it is needed to complete an optimization operation. The speedups gained are 1.84 for 2 processes, 3.36 for 4 processes, 5.56 for 8 processes and 9.27 for 16 processes. It is concluded that although the results are good, their implementations of relatively simple operations are cumbersome and heavy due to load balancing (repartitioning of the mesh) and synchronization between the submeshes. A parallel method for generating Delaunay triangulations has been explored in [3] where a mesh is decomposed into submeshes such that the insertion in one submesh will not propagate to other submeshes. After decomposing the mesh, an existing software library, Triangle [11], is used to do the actual mesh refinement on each submesh in parallel. While Triangle facilitates the use of a user-defined function for deciding which triangles should

be considered “big”, it did not have functionality for determining which triangles are of low quality in terms of the circumradius-to-shortest edge ratio or minimal angle so they had to reprogram and recompile Triangle.

In [9], a mesh based algorithm, called deformable simplicial complex (DSC), for deformable interface tracking in 2D and 3D is introduced. The interface (a curve in 2D and a surface in 3D) is represented explicitly as a piecewise linear curve or surface. The domain is also discretized by triangles in 2D and tetrahedra in 3D. The interface is then represented as the set of edges separating triangles marked as outside from those marked as inside. This approach has the advantage of space adaptivity, preservation of sharp details and control of topology. In [8] it is concluded that the implementation of DSC is difficult to use for new applications because it has been created with mainly fluid simulations in mind. Additionally, it does not support parallelized computations. In this thesis, these problems are addressed to make the framework more general and configurable, and parallelism is considered as well.

## 1.3 Overview of Thesis

Simulations as a concept is introduced in Section 2. A description of the requirements for what the implementation of the simulation framework should include, can also be found here. In Section 3, the theoretical background for simplicial complexes can be found. Especially Section 3.1 is important because it introduces the tools that are necessary for understanding how to read the algorithms for the mesh operations later in Section 7. Section 4 is a description of how a simplicial complex is implemented using a computational mesh. It also includes additional functions that are used for algorithms and, furthermore, an overview of the mesh operations and how to measure the quality of a mesh is given. In Section 5, the deformable simplicial complex is introduced. The steps of the algorithm is explained and the algorithm is presented. In Section 6 it is explained how a mesh is divided into submeshes. An algorithm for the slabbing method and the parallel deformable simplicial complex is given. It also explains how each mesh operation is restricted regarding the boundary between two submeshes. This is used in Section 7 which contains the algorithm of each mesh operation. It uses the topology algebra, functions and the submesh boundary restrictions from previous sections. Section 8 contains an explanation of how the concrete implementation has been made and Section 9 contains a description of the experiments that are done to ensure that the implementation is correct. Section 10 lists the results of the experiments and a discussion of the results can be found in Section 11. A conclusion of the entire project can be found in Section 12 and a list of suggestions for improvements and further work can be found in Section 13

## 1.4 Notation Summary

An overview of the notation used in this thesis is shown in Table 1.1.

| | |
|---|---|
| $\sigma$ | simplex of arbitrary dimension |
| $\sigma_k$ | simplex of dimension $k$ |
| $v_0, v_1, \ldots$ | vertices (0-simplices) |
| $e_0, e_1, \ldots$ | edges (1-simplices) |
| $t_0, t_1, \ldots$ | triangles (2-simplices) |
| $\langle v_0, \ldots, v_k \rangle$ | $k$-simplex spanned by vertices $v_0, \ldots, v_k$ |
| $[v_0, \ldots, v_k]$ | oriented $k$-simplex spanned by vertices $v_0, \ldots, v_k$ |
| $\Sigma, \{\sigma, \ldots\}$ | unordered sets of simplices |
| $\mathsf{vert}(\sigma)$ | set of vertices of simplex $\sigma$ |
| $V, W$ | sets of vertices |
| $E$ | set of edges |
| $T$ | set of triangles |
| $\lvert\Sigma\rvert$ | number of elements in simplex set $\Sigma$ |
| $K$ | simplicial complex |
| $\textsf{\textbf{boundary}}(\sigma)$ | boundary of simplex $\sigma$ |
| $\textsf{\textbf{boundary*}}(\sigma)$ | full boundary of simplex $\sigma$ |
| $\textsf{\textbf{star}}(\sigma)$ | star of simplex $\sigma$ |
| $\textsf{\textbf{star}}(\Sigma)$ | star of simplex set $\Sigma$ |
| $\textsf{\textbf{closure}}(\sigma)$ | closure of simplex $\sigma$ |
| $\textsf{\textbf{closure}}(\Sigma)$ | closure of simplex set $\Sigma$ |
| $\textsf{\textbf{link}}(\sigma)$ | link of simplex $\sigma$ |
| $\textsf{\textbf{skeleton}}_k(\Sigma)$ | $k$-skeleton of simplex set $\Sigma$ |
| $\textsf{\textbf{filter}}_k(\Sigma)$ | $k$-filter of simplex set $\Sigma$ |
| $\mathbf{v}, \mathbf{p}$ | vectors of arbitrary dimension |
| $\mathbf{v}^\perp$ | perpendicular vector of $v$ in two dimensions |
| $Q$ | quality measure |
| $Q^\Sigma$ | quality measure for simplicial set $\Sigma$ |
| $\textsc{Name}(\text{args})$ | function or algorithm |

**Table 1.1:** An overview of notation.

# 2 Simulations

A simulation is an imitation of a real-world system or process that changes over time. Simulations use a model to represent the state of the system and to describe the behaviour of it over time. Simulations are used in many fields of science and they can be useful on the entire size scale, from elementary particles to planets of the universe. Deforming curves and surfaces occur in everyday situations, e.g. the surface of water, clothes, glass blowing and clay modelling. In this project, two-dimensional curves will be the main focus, but some ideas also applies to surfaces in three dimensions. A curve is called an *interface* if it separates two regions that can be distinguished by a property (e.g. material). An example of an interface is a water surface, as it separates water from air or other phases, e.g. oil. A piece of cloth is not an interface, as it does not separate two different regions. If the region defining the interface is bounded, the interface is called *closed* or *watertight*. The regions that the interface separates are called *phases* and if the interface is bounded, the bounded region is called the *interior* and the outside region is called the *exterior*.

## 2.1 Simulation Framework

Multiple simulations (or other types of applications) fall into the group of closed interfaces, and having a generic framework that can be used for all these is appealing because the amount of work for each application will be reduced to the application specific behaviour. One such framework can be built around a *Deformable Simplicial Complex* (DSC). More about DSC can be found in Section 5. Figure 2.1 shows the layers of such a framework. The top layer is the application, which could be a liquid simulation, a sculpting tool or something else. Examples of simple applications are given in Section 9. The second layer is the DSC algorithm. This layer makes topological changes to the mesh using mesh operations described with high level simplex relations and other functions. Deformable Simplicial Complex is explained in Section 5 and mesh operations are explained in Section 4.2. The third layer, Simplex, is the implementation of the simplex relations and operations on top of a data structure (fourth layer). The simplex layer and the data structure layer are tightly coupled as the simplex layer must know how to navigate in the specific mesh data structure. Simplex relations and operations are defined in Section 3.1. The fifth and lowest layer is the hardware layer. A typical computer has units with support for parallel computations, and taking advantage of those is beneficial for the computational time. CPUs tend to get more cores and With CUDA and OpenCL, the graphics processing unit (GPU) is being used for general purpose computing. As indicated in Figure 2.1, one could gain the benefits of the hardware of the host computer by changing the data structure layer and the simplex layer while not changing anything in the application or the DSC layer.

The communication between the application and DSC is repeating back and forth until the simulation is done. The application computes a velocity field which, given a time step, leads to new positions (a displacement) of the interface, and this information is passed on to DSC. DSC updates the mesh and gives back the result to the application which computes a new velocity field, see Figure 2.2.

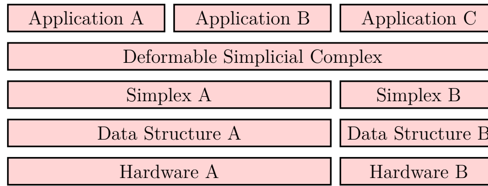



**Figure 2.1:** An example of the layers of the simulation framework.

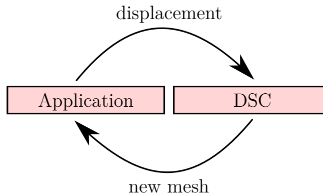
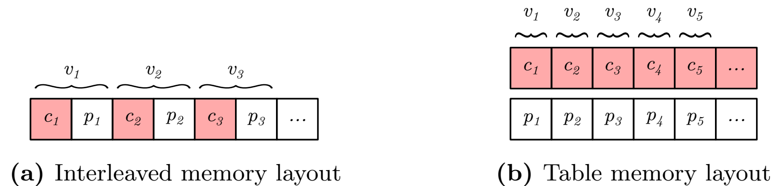



**Figure 2.2:** The interaction between DSC and the application.

## 2.2 Requirements

A previous project in [8] has analysed the current implementation of DSC and shown problems that will be addressed in this thesis. Some of the problems are design choices and some are about practical approaches to software development. The following is a description of the most important requirements to the new implementation.

**Consecutive data.** In previous implementations, the data of the simulation would be saved as data on the vertices (or edges or triangles) themselves, but this means that data of the same type is interleaved in memory. This makes it difficult to move it and, furthermore, algorithms that work on the same data will have more cache misses because spatial locality is less optimal [1]. The proposed solution is to arrange data of the same type in a table consecutively and then the data of a vertex is resolved by look up. If there is multiple data for each vertex (e.g. coordinates, velocity, pressure etc.), a table will exist for each type. See the example in Figure 2.3.




**(a)** Interleaved memory layout **(b)** Table memory layout

**Figure 2.3:** Examples of memory layouts. Each vertex $v_i$ has values $c_i$ (red) and $p_i$ (white). Figure 2.3a has less optimal spatial locality when an algorithm iterates over the same type of data. Figure 2.3b solves the problem by arranging data in separate tables. Spatial locality has influence on the effectiveness of memory caching.

**Parallelization.** When the 2D version of DSC was created, parallelism was not a priority, and as concluded in [8], the 3D version needs more work to make

the shift possible. In this project, the DSC algorithm is built from scratch and parallelism is considered from the beginning.

**Modularity.** Building the framework with modularity in mind means that certain parts and methods can be replaced without affecting other parts. This makes it easier to change to and experiment with other types of hardware, data structures and algorithms. The current implementations are difficult to change because they have been created with fluid simulations in mind. This means that the steps done by the algorithm is hard-wired to work best with the fluid simulation and it is not possible to change this without changing the framework itself.

**Type parametrization.** One of the improvements done in [8] to the DSC implementation in 3D is to free it from being dependent on the geometry library *GEometry and Linear algebra* (GEL)[1] and instead make the choice available to the user. The hard-wiring of GEL was a problem because it contained the real and vector types and this made it impossible to switch between e.g. double and float or change the vector implementation from GEL to OpenTissue, Boost or something else. Because DSC can be used by multiple types of applications, it will also be used by different people who have different opinions about external dependencies. The 3D version of DSC is now mostly type parametrized, but on the lowest levels there are still dependencies and this project will be designed with type parametrizations from the start.

**Adjustable quality measures.** The quality measures used for mesh refinement, coarsening etc. must be replaceable and the threshold values for the default quality measures must be easily configurable from the application because these often requires fine-tuning for the specific application.

**Interpolators.** When the mesh is updated, the data of the components of the mesh might change. It must be possible to change the functions that generates values for these updates. Often interpolators are used to compute a new value from neighbour values. Many interpolation methods exist, and the preferred one depends on the application. The implementation in this project should support custom interpolators.

**Multiplatform support.** The 3D version of DSC has been developed on the same machine for a long time, and some of the platform specific project files were outdated. This was remedied in [8] by introducing CMake[2] which is a tool for creating project files from platform independent configuration files. This means that project files do not have to be updated and maintained manually and new developers can get automatically created project files for his favoured platform.

**Code organization.** In the 3D version of DSC, the application code and the DSC framework was combined into the same project. This made it difficult

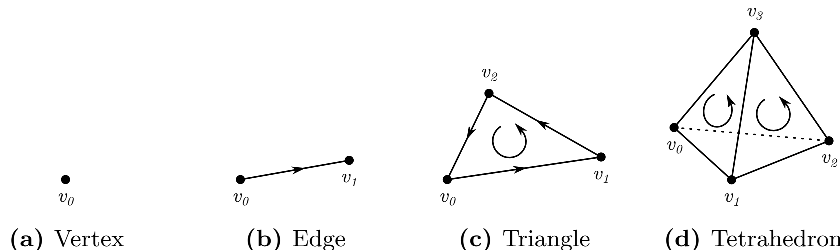
[1] GEL website: `http://www2.imm.dtu.dk/projects/GEL`
[2] CMake website: `http://www.cmake.org`

to get an overview of what was shared library code and what was framework code. Furhermore, some compilers do not accept multiple `main` functions in the same project. This was changed in [8] by creating multiple projects: one for the framework, and one for each application. This approach is also taken in this project to ensure separation and clarity about what is framework and what is application.

# 3 Simplicial Complexes

The following is a description of the mathematical notions that serve as the theoretical background the for implementation of the deformable simplicial complex algorithm. Especially the simplex relations and operations defined in Section 3.1 are necessary to understand the implementations of the mesh operations in Section 7, although the introduction of them in Section 4.2 does not require this understanding.

In mathematics, a simplicial complex is a set of components called *simplices* (*simplex* in singular) which represents a topological space. A $k$-simplex $\sigma_k$ in $\mathbb{R}^d$, $k \leq d$ is the convex hull of $k+1$ affinely independent points $\mathsf{vert}(\sigma_k) = \langle v_0, v_1, v_2, \ldots, v_k \rangle$, where the points $v_0, v_1, v_2, \ldots, v_k$ are called the vertices of the simplex. See Figure 3.1 for an example. An oriented simplex is a simplex where the ordering of the vertices is significant. Oriented simplices are notated with its vertices in square brackets, i.e. $[v_0, v_1, v_2, \ldots, v_k]$. A set of simplices is denoted by $\Sigma$. The terms vertex, edge and triangle and 0-simplex, 1-simplex and 2-simplex will by used interchangeably. A simplex $\sigma$ is a *face* of simplex $\sigma'$ if $\mathsf{vert}(\sigma) \subseteq \mathsf{vert}(\sigma')$. For example, triangle $[v_0, v_1, v_2]$ in Figure 3.1c has the faces $[v_0]$, $[v_1]$, $[v_2]$, $[v_0, v_1]$, $[v_1, v_2]$, $[v_2, v_0]$ and $[v_0, v_1, v_2]$. Note that any simplex $\sigma$ is a face of itself because $\mathsf{vert}(\sigma) \subseteq \mathsf{vert}(\sigma)$. A subface of a simplex $\sigma$ is a face which has lower dimensionality than $\sigma$. E.g. the edge $[v_0, v_1]$ and the vertex $[v_0]$ in Figure 3.1c are subfaces of the triangle $[v_0, v_1, v_2]$. A superface of a simplex $\sigma$ is a simplex that has $\sigma$ as a subface. E.g. the triangle $[v_0, v_1, v_2]$ in Figure 3.1c is a superface to the edge $[v_0, v_1]$ and the vertex $[v_0]$.




**(a)** Vertex **(b)** Edge **(c)** Triangle **(d)** Tetrahedron

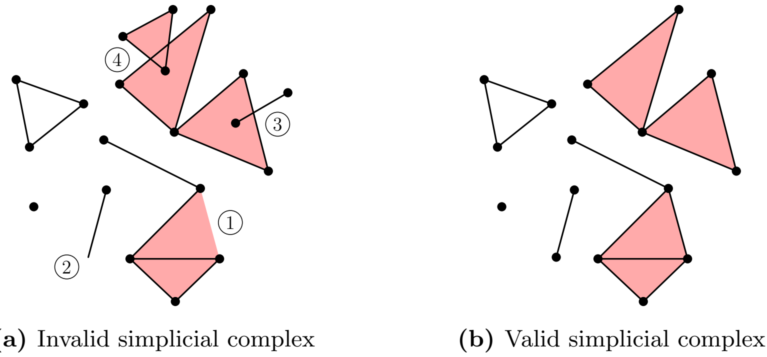
**Figure 3.1:** Oriented $k$-simplices with $0 \leq k \leq 3$. The arrows show the orientation.

A simplicial complex $K$ is a set of simplices where the following two conditions hold:

1. For each simplex $\sigma \in K$, all faces of $\sigma$ are also contained in $K$.

2. The intersection $\sigma \cap \sigma'$ of two simplices $\sigma, \sigma' \in K$, is one or more faces of both $\sigma$ and $\sigma'$ or the empty set.

Figure 3.2 is an example of an invalid and a valid simplicial complex. In Figure 3.2a, consider ① and ② where faces are missing and thus the first condition for being a simplicial complex is not satisfied. Additionally, ③ and ④ violates the second condition because the intersections are not simplices in the simplicial complex. Figure 3.2a is therefore not a simplicial complex. Figure 3.2b is a valid simplicial complex because both conditions are satisfied.




**(a)** Invalid simplicial complex **(b)** Valid simplicial complex

**Figure 3.2:** Examples of an invalid and a valid simplicial complex. 0-simplices are shown as black dots and 1-simplices as black lines. Red colour indicates 2-simplices (triangles). Figure 3.2a is an invalid simplicial complex because of missing faces at ① and ②. At ③ and ④, additionally, there are intersections which are not faces of the simplicial complex. Figure 3.2b has no missing faces or intersections and is a valid simplicial complex.

## 3.1 Simplex Relations and Operations

Navigating and obtaining information about a simplicial complex is the purpose of simplex relations and operations. The relations and operations are expressed using set theory, and some may be expressed as compositions of other relations or operations. Relations are defined on simplices only, while operations are also defined on sets of simplices. Let **op** be a simplex operation on a set of simplices $\Sigma$, then the result is the union of the operation on each simplex in $\Sigma$, such that

$$\textsf{\textbf{op}}(\Sigma) = \bigcup_{\sigma \in \Sigma} \textsf{\textbf{op}}(\sigma).$$

The result of some simplex operations are valid simplicial complexes, while others are not. The following is a description of the most important operations and relations as defined in [9].

**Boundary.** The boundary of a simplex $\sigma$ is the set of all simplices in the simplicial complex $K$ which are a face of $\sigma$ and have dimensionality exactly one lower than $\sigma$, defined by

$$\textsf{\textbf{boundary}}(\sigma_k) = \{\sigma_{k-1} \in K \mid \textsf{vert}(\sigma_{k-1}) \subset \textsf{vert}(\sigma_k)\}.$$

Figure 3.3 shows an example of the boundary relation.

**Full boundary.** The full boundary of a simplex $\sigma$ is the set of all simplices in the simplicial complex $K$ which are a face of $\sigma$ and have lower dimensionality than $\sigma$, defined by

$$\textsf{\textbf{boundary*}}(\sigma_k) = \{\sigma_n \in K \mid \textsf{vert}(\sigma_n) \subset \textsf{vert}(\sigma_k)\}.$$

Note that in the definition above we have $n < k$ because we use a proper subset. Figure 3.4 shows an example of the full boundary relation.

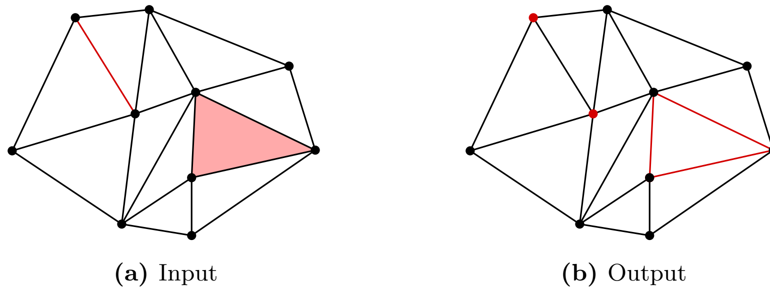



**Figure 3.3:** Example of the boundary relation. Input simplices are shown in red in Figure 3.3a. Output simplices are shown in red in Figure 3.3b. The boundary of an edge is the vertices at its end points and the boundary of a triangle is the edges of the triangle (but *not* the vertices).

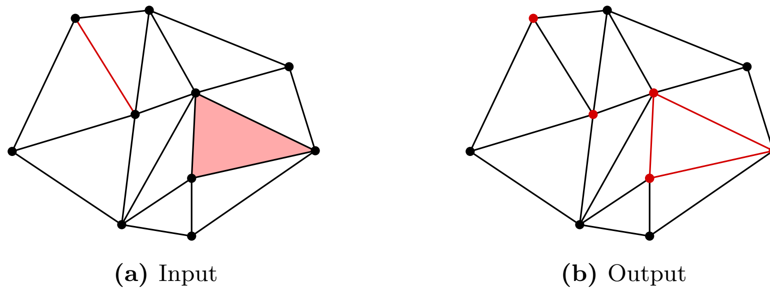



**Figure 3.4:** Example of the full boundary relation. Input simplices are shown in red in Figure 3.4a. Output simplices are shown in red in Figure 3.4b. The full boundary of an edge is the vertices at its end points and the full boundary of a triangle is the edges and vertices of the triangle.

**Star.** The star of a simplex $\sigma$ is the set of all simplices in the simplicial complex $K$ which have $\sigma$ as a face, defined by

$$\textbf{star}(\sigma) = \{\sigma' \in K \mid \mathsf{vert}(\sigma) \subseteq \mathsf{vert}(\sigma')\}.$$

Figure 3.5 shows an example of the star operation.

**Closure.** The closure of $\sigma$ is $\sigma$ itself and all its subfaces, in other words, it is the union of the set $\{\sigma\}$ and the full boundary of $\sigma$, defined by

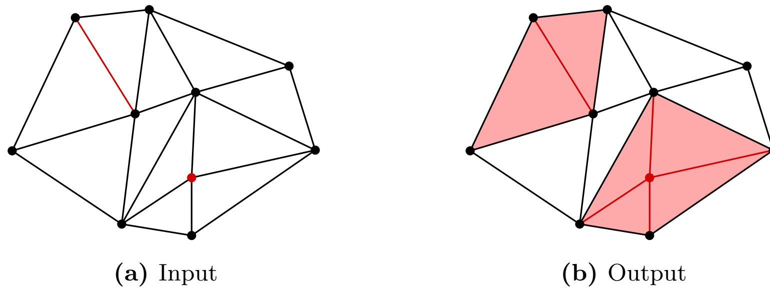
$$\textbf{closure}(\sigma) = \{\sigma\} \cup \textbf{boundary*}(\sigma).$$

Figure 3.6 shows an example of the closure operation.

**Link.** The link of a simplex $\sigma$ is expressed using two other operations, **closure** and **star**. The link of simplex $\sigma$ is the simplices in the closure of the star of $\sigma$, which do not have $\sigma$ as a face, defined by

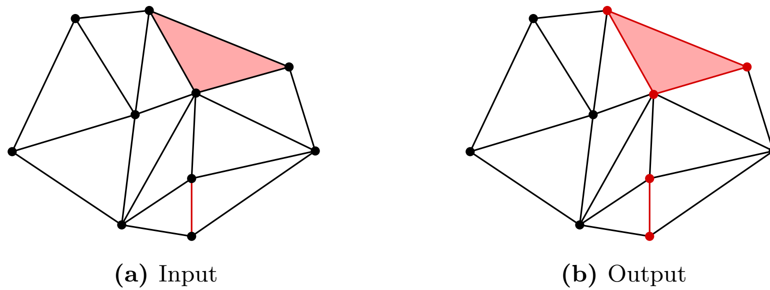
$$\textbf{link}(\sigma) = \textbf{closure}(\textbf{star}(\sigma)) \setminus \textbf{star}(\textbf{closure}(\sigma)).$$

Figure 3.7 shows an example of link.



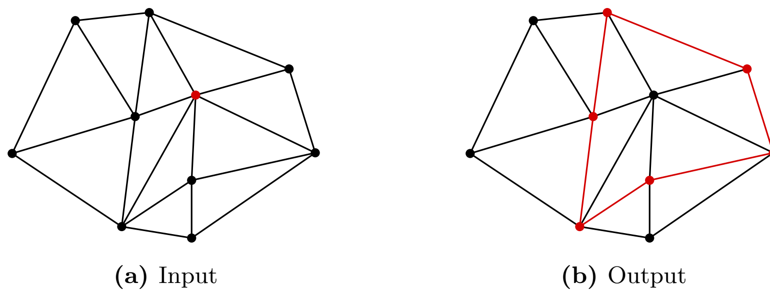
**(a)** Input **(b)** Output

**Figure 3.5:** Example of star. Input simplices are shown in red in Figure 3.5a. Output simplices are shown in red in Figure 3.5b. The star of a vertex is the vertex itself (because a simplex is a face of itself) and all the triangles and edges that have the vertex as a face. The star of an edge is the edge itself and the (up to) two triangles which has the edge as a face.



**(a)** Input **(b)** Output

**Figure 3.6:** Example of closure. Input simplices are shown in red in Figure 3.6a. Output simplices are shown in red in Figure 3.6b. The closure of an edge is the edge itself and the vertices of its end points. The closure of a triangle is the triangle itself, its edges and its vertices.



**(a)** Input **(b)** Output

**Figure 3.7:** Example of link. Input simplices are shown in red in Figure 3.7a. Output simplices are shown in red in Figure 3.7b.

**Skeleton.** The $k$-skeleton of a simplex set $\Sigma$ is the set of all faces which are of dimensionality $k$ or lower, defined by

$$\textsf{\textbf{skeleton}}_k(\Sigma) = \{\sigma_n \in \Sigma \mid n \leq k\}.$$

Figure 3.8 shows an example of $k$-skeleton.

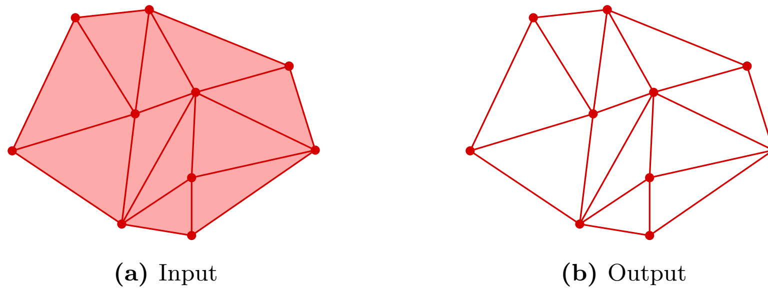



**Figure 3.8:** Example of the 1-skeleton. Input simplices are shown in red in Figure 3.8a. Output simplices are shown in red in Figure 3.8b. The output is the set of all simplices of dimension 1 or lower, that is, all edges and vertices, but no triangles.

**Filter.** The $k$-filter of a simplex set $\Sigma$ is the set of all faces which are of dimensionality exactly $k$, defined by

$$\textsf{\textbf{filter}}_k(\Sigma) = \{\sigma_n \in \Sigma \mid n = k\}.$$

Figure 3.9 shows an example of the filter.

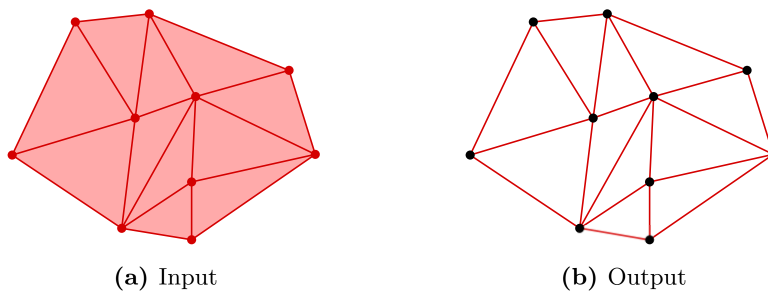



**Figure 3.9:** Example of 1-filter. Input simplices are shown in red in Figure 3.9a. Output simplices are shown in red in Figure 3.9b. The output is the set of all edges (1-simplices) but none of the vertices or triangles.

To summarize, boundary and full boundary are relations which means that they are used on simplices but not simplex sets. Star, closure and link are operations which can be used on both simplices and simplex sets. Skeleton and filter are used on simplex sets.

## 3.2 Discrete Manifold

An $n$-dimensional discrete manifold is an $n$-dimensional simplicial complex which satisfies two properties:

1. For each simplex $\sigma$ *not* on the boundary, the union of all incident $n$-simplices forms an $n$-dimensional ball (disc in two dimensions).

2. For each simplex $\sigma$ on the boundary, the union of all $n$-dimensional incident $n$-simplices forms a circular sector.

Figure 3.10 contains examples of invalid and valid discrete manifolds. In Figure 3.10a, consider ① where a "dangling" 1-simplex (edge) is on the boundary, but a circular sector formed by the union of the incident 2-simplices does not exist and the second property of a discrete manifold is not satisfied. In Figure 3.10b, consider ② which is not a boundary simplex. At any point inside the simplex, a disc can be drawn (first property of a discrete manifold). At ③ and ④, which are boundary simplices, circular sectors can be drawn (second property of a discrete manifold) and thus both properties are satisfied.

**(a)** Invalid manifold

**(b)** Valid manifold

**Figure 3.10:** Examples of an invalid and valid discrete manifold. 0-simplices are shown as black dots, 1-simplices as black lines and 2-simplices as red triangles. The red dots shows examples of points where discs or circular sectors are drawn using dashed lines in black. Figure 3.10a is an invalid manifold because a disc at ① does not exist. Figure 3.10b is a valid manifold. ② is an example of a full disc while ③ and ④ are examples of circular sectors on boundary simplices.

# 4 Mesh

A mesh is a collection of vertices, edges and faces that together defines a polyhedral object in three dimensions or a polygon in two dimensions. In this thesis, we are restricted to two dimensions. A mesh is often used as a discrete approximation of a continuous domain and discretized values can be stored in vertices, edges or faces. The finite element method (FEM) is an example of a numerical technique for finding approximate solutions to boundary value problems for differential equations. The elements referred to can be bound to the components of a mesh. A mesh can be used to represent a simplicial complex where 0-simplices are vertices in the mesh, 1-simplices are edges and 2-simplices are triangles in the mesh. If a mesh implementation supports polygons (faces with more than 3 corners), these are not considered because they do not exist in a simplicial complex. The following is a description of the functions (in addition to the simplex relations and operations introduced in Section 3.1) that a mesh must implement. These functions are used to implement the mesh operations described in Section 4.2. The quality of a mesh is discussed in Section 4.3.

## 4.1 Mesh Functions

In contrast to the simplex relations and operations which are used for navigating and querying a simplicial complex, mesh functions are used to modify the simplicial complex. The following is a list of the most important functions that a mesh must implement.

| | |
|---|---|
| $\sigma_0$ Insert($K$) | Insertion of 0-simplex in $K$ |
| $\sigma_2$ Insert($\langle\sigma_0, \sigma_0, \sigma_0\rangle, K$) | Insertion of 2-simplex and its boundary |
| void Remove($\Sigma$) | Removal of all simplices in $\Sigma$ |
| sign Orientation($\sigma$) | Orientation of simplex $\sigma$ |
| bool IsSubmeshBoundary($\sigma$) | Is $\sigma$ is on the submesh boundary? |
| bool IsInterface($\sigma$) | Is $\sigma$ is on the interface? |
| bool IsBoundary($\sigma$) | Is $\sigma$ is on the (mesh) boundary? |
| bool IsValidSimplex($\sigma$) | Is $\sigma$ is a valid simplex? |
| id Phase($\sigma$) | Phase of simplex $\sigma$ |

Note that the insertion of a 2-simplex does not require it to be in a specific orientation.

## 4.2 Mesh Operations

Mesh operations change the mesh by adding, deleting or moving elements. Some operations refine the mesh by adding more elements to increase the level of detail, while others may remove elements to decrease the level of detail. Some operations work to improve the shape of triangles. Mesh operations are used in conjunction with a quality measure as explained in Section 4.3. An overview of the mesh operations is given in Table 4.1. The following is a description of different mesh operations with an explanation of how they work and what their intended purpose is. Note that this is just a subset of all the possible mesh operations.

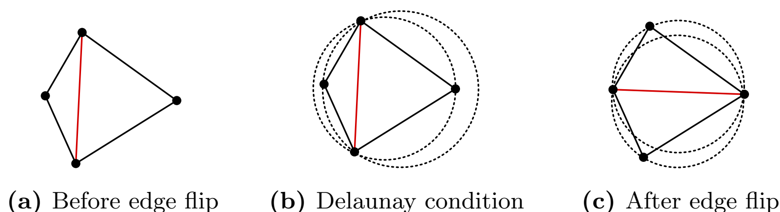
| | |
|---|---|
| Edge flip | Used for optimization |
| Loop subdivision | Used for refinement |
| Local retriangulation | Used for coarsening (or optimization) |
| Edge split | Used for refinement |
| Edge collapse | Used for coarsening |

**Table 4.1:** An overview of mesh operations.

**Edge flip.** If an edge is a subface of two triangles, it can be "flipped" such that its end points are the two vertices in the triangles which were not end points of the edge before the flip. If the edge is on the boundary of the mesh, it cannot be flipped because it is not a subface by two triangles. As shown in Figure 4.1, edge flip can produce invalid geometry and in these cases, edge flip is illegal. Edge flipping is used in Delaunay triangulations where edges are flipped if and only if the circumcircle of the three points of the first triangle contains the fourth point of the second triangle [5]. This is called the Delaunay condition, see the example in Figure 4.2. Using the Delaunay condition, we can to some extend avoid thin slivers and produce more regular triangles.

**(a)** Before edge flip **(b)** Invalid geometry

**Figure 4.1:** Example of an illegal flip. Flipping the red edge in Figure 4.1a produces the invalid geometry in Figure 4.1b. Edge flip is in this example not legal.

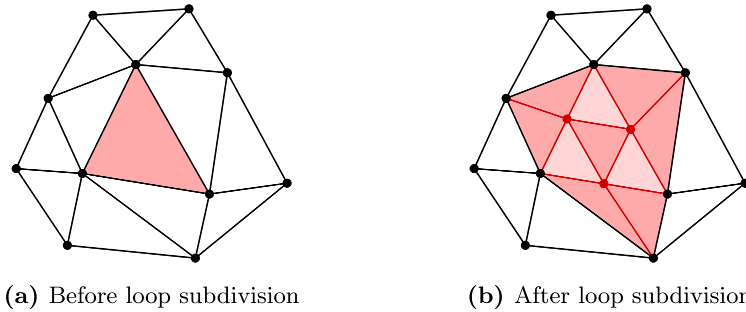


**(a)** Before edge flip **(b)** Delaunay condition **(c)** After edge flip

**Figure 4.2:** Example of edge flip and the Delaunay condition. Figure 4.2a is before the edge has been flipped. Figure 4.2b shows the Delaunay condition which is not satisfied because the fourth vertex that is left when drawing a circle around the three vertices of a triangle is inside the circle (for both triangles). Figure 4.2c is after the edge has been flipped. The Delaunay condition is now satisfied because the vertices are outside.

**Loop subdivision.** Adding triangles to a mesh refines it and makes it more detailed. Loop subdivision inserts a smaller triangle into a larger one, and to do so, it must split the three surrounding triangles into two triangles each, see the example in Figure 4.3. The new vertices of the smaller triangle are inserted

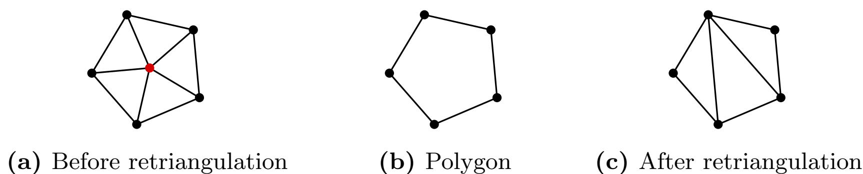
somewhere between the end points of the edge, e.g. halfway between them as in the example.



**(a)** Before loop subdivision **(b)** After loop subdivision

**Figure 4.3:** Example of loop subdivision. The input 2-simplex is shown in red in Figure 4.3a and the new simplices inserted after loop subdivision are shown in red in Figure 4.3b. A smaller triangle is inserted into the input triangle, causing the neighbouring triangles to be split.

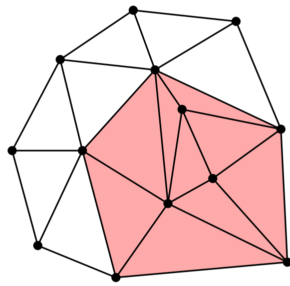
**Local retriangulation.** Removing triangles from a mesh makes it less detailed, and local retriangulation is an example of a coarsening mesh operation. By removing triangles, the remaining polygon (the trace of the removed triangles) can be retriangulated and the result will be a mesh with fewer triangles, see the example in Figure 4.5. Note that a triangulation of an $n$-polygon has exactly $n-2$ triangles [5], which means means that the chosen subset of simplices must contain at least one vertex (as in the example in Figure 4.5) which is "inside" the mesh and not a vertex of the polygon. Multiple retriangulation algorithms exist, e.g. the monotone piece method in [5] and ear clipping method in [10]. Note that some retriangulation methods can also be used to improve the quality of the triangles, e.g. a Delaunay triangulation will improve the quality given in Equation 4.



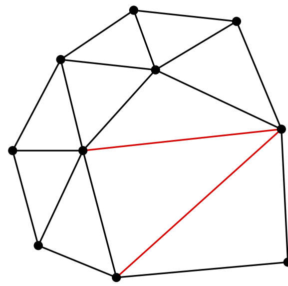
**(a)** Before retriangulation **(b)** Polygon **(c)** After retriangulation

**Figure 4.4:** Example of retriangulation with a single vertex inside the polygon. Figure 4.4a shows the simplices before the retriangulation. The red vertex is the one which inside the polygon. Figure 4.4b shows the polygon before retriangulation and the result is shown in Figure 4.4c. Observe that the final retriangulation has fewer triangles than the original.

**Edge split.** An edge can be refined by splitting it into two edges. This will also cause the triangles that have the edge as subface to be split, as seen in the example in Figure 4.6. The inserted vertex that can be placed anywhere between the two end points of the edge (not including the end points).

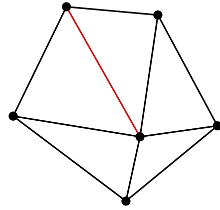


**(a)** Before local retriangulation

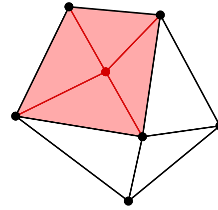


**(b)** After local retriangulation

**Figure 4.5:** Example of local retriangulation. The input 2-simplex is shown in red in Figure 4.5a and the new simplices inserted after local retriangulation is shown in red in Figure 4.5b. A subset of simplices is removed and the remaining polygon is retriangulated.



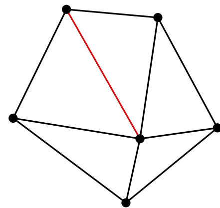
**(a)** Before edge split



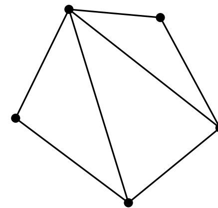
**(b)** After edge split

**Figure 4.6:** Example of edge split. The input 1-simplex is shown in red in Figure 4.6a and the new simplices inserted after edge split is shown in red in Figure 4.6b. The edge is split into two edges by inserting a vertex between the two end points and splitting the two triangles that have the edge as subface.

**Edge collapse.** A mesh can be coarsened by "shrinking" an edge which collapses the triangles that have the edge as subface. The point where the two end points collapses can be either at one of the original end points as shown in Figure 4.7 or somewhere in between.



**(a)** Before edge collapse



**(b)** After edge collapse

**Figure 4.7:** Example of edge collapse. The input 1-simplex is shown in red in Figure 4.7a and the result after edge collapse is shown in Figure 4.7b. The two end points are collapsed and the triangles which have the edge as subface are removed because they degenerate.

## 4.3 Quality Measures

For some models, the accuracy of an approximation represented by a mesh depends on the regularity of the mesh. The regularity can be quantified using quality measures. The global quality of the mesh is determined by the quality of each local component, that is, the quality of the vertices, edges and triangles. In other words, a quality measure takes a mesh component as input and returns a number that represents the quality. Each phase may have different quality measures because there can be different requirements for them depending on the application. Note that the interface consists only of vertices and edges, and it does therefore not make sense to have a triangle quality measure for that. The quality of a mesh is changed by using the mesh operations defined in Section 4.2. The following is a description of some of the quality measures for each type of mesh component.

**Vertices.** The simplest component of a mesh is a vertex. If a vertex is supposed to be moved, a quality measure for a vertex $v$ could be the distance between $v$ and the final destination $v'$ in the function

$$Q(v) = \operatorname{dist}(v, v') \tag{1}$$

where dist is some distance function.

**Edges.** Similar to the vertex quality measure, the length of an edge can be used as a quality measure for an edge $e$ in the function

$$Q(e) = \operatorname{length}(e) \tag{2}$$

where length is a function that calculates the length of the edge. Note that the squared length or some other length measure can also be used.

**Triangles.** Multiple types of quality measures exist for triangles because there are various properties to consider: edge lengths, angles and area. The area of the triangle $t$ can be used for controlling the size of the triangles in the function

$$Q(t) = \operatorname{area}(t) \tag{3}$$

where area is a function that calculates the area of $t$. [12] includes many different triangle quality measures that are more advanced, for example the function

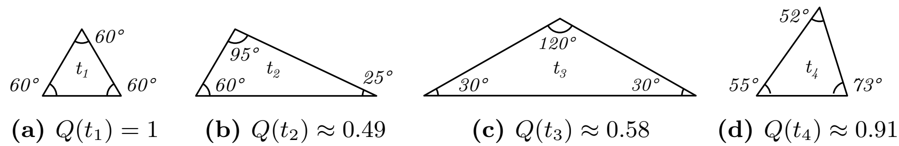
$$Q(t) = \frac{2}{\sqrt{3}} \sin \theta_{\min} \tag{4}$$

where $\theta_{\min}$ is the minimum angle in $t$. This quality measure is improved by a Delaunay triangulation.[3] The example in Figure 4.8 shows four triangles. Using the quality measure in Equation 4, the qualities are computed as

[3] Delaunay triangulation is explained in Section 4.2 under Edge flip.

$$
\begin{aligned}
Q(t_1) &= \frac{2}{\sqrt{3}} \sin 60^\circ = 1 \\
Q(t_2) &= \frac{2}{\sqrt{3}} \sin 25^\circ \approx 0.49 \\
Q(t_3) &= \frac{2}{\sqrt{3}} \sin 30^\circ \approx 0.58 \\
Q(t_4) &= \frac{2}{\sqrt{3}} \sin 52^\circ \approx 0.91
\end{aligned}
$$

This shows that this particular quality measure favours equilateral triangles for which the quality measure gives the value 1 and values going towards zero when triangles are going towards being less equilateral.




**(a)** $Q(t_1) = 1$ **(b)** $Q(t_2) \approx 0.49$ **(c)** $Q(t_3) \approx 0.58$ **(d)** $Q(t_4) \approx 0.91$

**Figure 4.8:** Examples of using a quality measure. The equilateral triangle in Figure 4.8a has a value of 1 and the almost equilateral triangle in Figure 4.8d is almost 1. The two triangles which are relatively far from being equilateral in Figure 4.8b and Figure 4.8c have values closer to zero.

# 5 Deformable Simplicial Complex

In a deformable simplicial complex (DSC), the closed interface is represented by a piecewise linear curve in 2D (or a triangulated surface in 3D) [9]. The domain, which is usually a box containing the interface, is triangulated (tetrahedralized in 3D) such that the interface is edges (or triangles) in the triangulation. The triangulation must fulfill the conditions of the simplicial complex. Because DSC is restricted to closed interfaces, the triangles (or tetrahedra) can be divided into interior and exterior triangles. The interface is then given implicitly as the set of simplex faces that divide the interior from the exterior, see the example in Figure 5.1. Displacements of the vertices of the interface deforms the interface. When doing so, the simplicial complex condition might be violated when triangles intersect. DSC handles this by using local mesh optimization algorithms, see the example in Figure 5.2. The interface may inclose multiple disjoint interior parts, where each is called a phase. Two phases may or may not be of the same type. For example a water droplet is of the same type of phase as the water it emitted from, but a system with both water and oil will have two different phases.

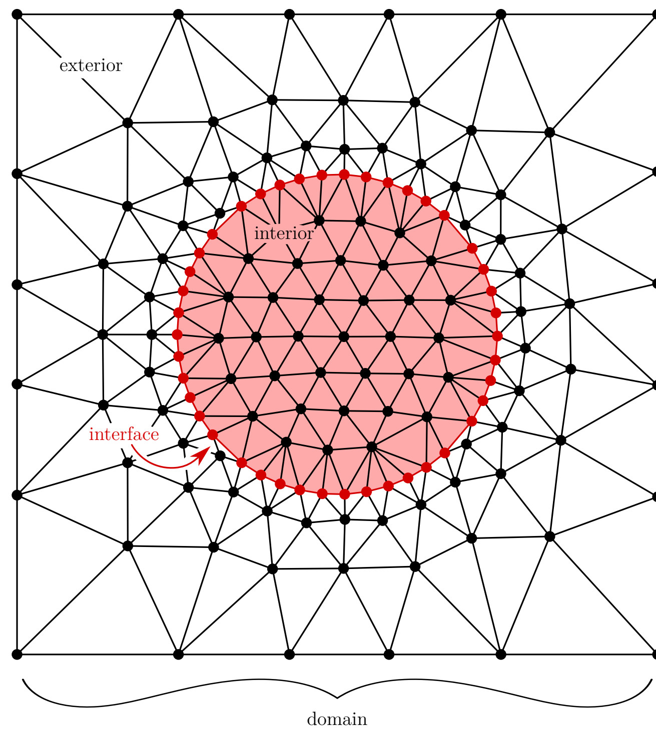



**Figure 5.1:** An example of a deformable simplicial complex mesh. The exterior consists of white 2-simplices and the interior of red 2-simplices. The set of red edges and vertices is the interface given implicitly as the simplices that divide the interior from the exterior.

This method has several benefits compared to other methods [9]. It has control of the topology of the interface which can change automatically when collision occurs, but also preserve it. It can represent details of different magnitude and

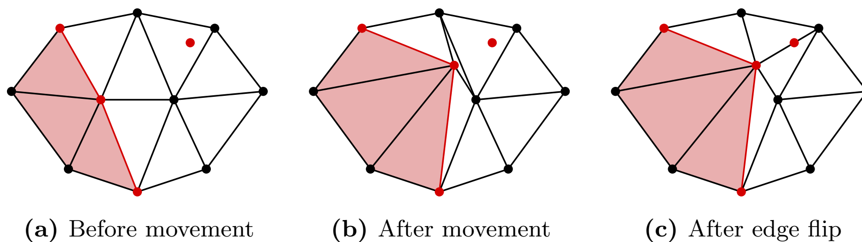
preserve sharp details that are lost in some methods, e.g. grids. The DSC algorithm consists of multiple steps that may be repeated a number of times for each time step in the simulation. Each step is performed separately for all the phases and the interfaces between them. The following is a description of all the steps.

## 5.1 Movement

A physics simulation calculates a velocity field which describes the motion of each a vertex from time $t$ to $t+1$, for example by using the explicit forward Euler step, defined as

$$\mathbf{p}_{t+1} = \mathbf{p}_t + \Delta t \mathbf{v}_t. \tag{5}$$

It may not be possible to move the vertices all the way because the changed 2-simplices (triangles) connected to them can create intersections that violates the simplicial complex definition. If the vertices are moved as far as possible without creating intersections, the mesh optimization steps can be used to make it possible for the vertices to move even further, see an example in Figure 5.2. By repeating this, the vertices will approach the new position.



**(a)** Before movement **(b)** After movement **(c)** After edge flip

**Figure 5.2:** A sample of a mesh with two phases, red and white. The unconnected red dot is the target position for the middle vertex on the red interface between the two phases. Figure 5.2a shows the mesh before movement. Figure 5.2b is the result after moving the vertex as far as possible without inverting triangles. Figure 5.2c shows the sample mesh after flipping an edge. The vertex can now move to the target without inverting any triangles.

## 5.2 Coarsening

When the triangulation of a subset of a domain becomes more detailed than necessary, the mesh is coarsened which means that triangles in that subdomain are removed and the remaining triangles are made larger. This is advantageous because less time and memory spend in regions of less interest means that more time and memory can be spend in regions of interest. Examples of mesh operations that coarsen a mesh are local retriangulation and edge collapse.

## 5.3 Refinement

If a subset of the domain needs to be more detailed perhaps because there are many changes and finer details to be represented, the refinement procedure creates more triangles. This will also make the computational time and memory

impact larger, so it is important to correctly identify areas where refinement is needed. Examples of mesh operations that refine a mesh are loop subdivision and edge split.

## 5.4 Optimization

Even when the size of the triangles (or edges) in a subset of the domain is optimal, the shape of the triangles may not be so. Section 4.3 shows an example of a triangle quality measure that uses the angles and favours equilateral triangles. The optimization procedure should optimize the mesh according to this (or a similar) quality measure. An example of a mesh operation that can be used to optimize the mesh is edge flip. Some triangulation implementations may also improve the quality, e.g. Delaunay triangulation.

## 5.5 Smoothing

The smoothing step moves vertices to optimize the shape of the triangles, but it does not add or remove vertices. An example of a smoothing method is Laplacian smoothing, defined as

$$\mathbf{p}_i = \frac{1}{N}\sum_{j=1}^{N}\mathbf{p}_j, \tag{6}$$

where $\mathbf{p}_i$ is the new position of the $i$'th vertex (which is not a vertex on the boundary of the domain or an interface vertex) and $\mathbf{p}_j$ for $j = 1 \ldots N$ are the positions of the $N$ vertices that $p_i$ is connected to. This update can be repeated until some stop criterion is met. The boundary of the domain and the interface can also be smoothened, but these should preserve the shape of the domain and interface. Note that if the vertices contain data that depends on their position, new values will have to be created for these (e.g. by interpolation). Due to the time constraint, mesh smoothing has not been implemented.

## 5.6 Algorithm

The deformable simplicial complex algorithm can be boiled down to two repeating steps:

1. Move all vertices as far as possible (movement).
2. Optimize mesh (coarsening, refinement, optimization, smoothing).

These steps are repeated until all vertices are moved or some other criterion makes it stop. Because the interface and each phase may have different quality measures, these are considered independently of each other. A quality measure for phase $p$ during refinement is notated as $Q^p_{\text{refinement}}$. $Q^K$ is the quality measure used as stop criterion (e.g. all vertices are moved, a maximum number of iterations has been reached etc.). The algorithm is shown in Algorithm 5.1. First the phases and the interface of the simplicial complex $K$ are found. The main loop of the algorithm uses $Q^K$ to determine whether the algorithm is done or not. If not, the movement procedure is called which moves the vertices towards their destinations as determined by the velocity field computed

by the application. After that, the interface is coarsened and refined by calling MeshUpdate in Algorithm 5.2. In this procedure, a strategy for selecting simplices of low quality is used. This strategy must have defined HasNext, which returns whether there are more simplices to update, and GetNext which returns the next simplex that must be updated. The mesh operation used is given as the parameter Op. At last, the phases are updated with coarsening, refinement and optimization mesh operations. The steps and mesh operations used in Algorithm 5.1 are examples and they can be replaced by other steps or operations depending on the application.

**Algorithm 5.1** Deformable Simplicial Complex

**Input:** A simplicial complex $K$.

```
procedure DSC(K)
    P ← Phases(K)
    I ← Interface(K)

    while Q^K is too low do

        Movement(Q^K_movement, K)

        MeshUpdate(I, Q^I_coarsening, EdgeCollapse, K)
        MeshUpdate(I, Q^I_refinement, EdgeSplit, K)
        MeshUpdate(I, Q^I_smoothing, InterfaceSmoothing, K)

        for all p ∈ P do
            MeshUpdate(p, Q^p_coarsening, LocalRetriangulation, K)
        end for

        for all p ∈ P do
            MeshUpdate(p, Q^p_refinement, LoopSubdivision, K)
        end for

        for all p ∈ P do
            MeshUpdate(p, Q^p_optimization, EdgeFlip, K)
        end for

        for all p ∈ P do
            MeshUpdate(p, Q^p_smoothing, LaplacianSmoothing, K)
        end for

    end while
end procedure
```

**Algorithm 5.2** Mesh Update Procedure

**Input:** A simplex set $\Sigma$, a quality measure $Q$, a mesh operation Op and a simplicial complex $K$.

```
procedure MeshUpdate(Σ, Q, Op, K)

    while HasNext(Q, Σ) do
        s ← GetNext(Q, Σ)
        Op(s, K)
    end while

end procedure
```

# 6 Mesh Decomposition

The mesh operations all work by only optimizing the mesh locally, and therefore the computational time can be decreased by performing those on the mesh at multiple locations simultaneously. They must be performed such that two operations cannot change the same simplex at the same time, and since the operations works on multiple simplices at once, it is reasonable to divide the mesh into smaller contiguous submeshes. For the submeshing, the following conditions must hold:

1. The union of the submeshes is the original mesh.
2. If two submeshes shares a boundary, the intersection of the submeshes is the vertices and edges of the shared boundary.
3. If two submeshes do not share a boundary, the intersection of the submeshes is the empty set.

The boundary between two submeshes will be referred to as *submesh boundary*. An example of a mesh decomposition method, or submeshing method, is the slabbing method where vertical or horizontal lines divide the mesh into submeshes. Another similar method is where the mesh is divided both vertically and horizontally to create blocks. A more complex method is one where $n$ points are placed randomly inside the mesh. A Voronoi diagram is then computed for these points and the simplices are assigned to the cell it is inside. The simplices of each cell becomes a submesh, and we get $n$ submeshes. Because this may create unbalanced submeshes, a spring-like system moves the boundaries until a state of equilibrium is found. The following is a description of how the submeshes are created and what restrictions there are on the mesh operations concerning the submesh boundary. The mesh operations are expressed using topological algebra in Section 7.

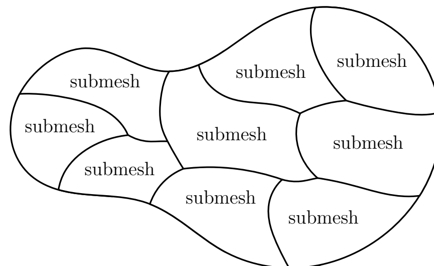



**Figure 6.1:** An example of a mesh divided into submeshes.

## 6.1 Red-Black Ordering

Traditional red-black ordering divides a mesh into "red" regions and "black" regions [13]. The regions must be interleaved such that no red region will be neighbour to another red region and the same for black regions. The regions will be computed in a two step procedure. First, all the red regions are computed simultaneously, and when they are done, the black ones are computed.

This is necessary for example when the computation of a node in a regular grid is dependent on the values of the neighbouring nodes. In this case, the operations on a submesh depends on a neighbouring submesh because a mesh operation does not only change the simplex it gets as input, it may also change the simplices in a local neighbourhood, e.g. loop subdivision splits the triangles that are adjacent to the input triangle. Thus, if the red-black scheme is used, when we are working on the black submeshes, we are allowed to change the boundary because it does not affect the other black submeshes. This assumes that the distance between two submeshes is large enough. If there is only one 2-simplex between them for instance, this does not hold. Additionally, in distributed memory systems where each process has its own local memory (e.g. the local memory of each SPU in a GPU), the simplices on the boundary would have to be synchonized and the implementation would have to include latency hiding and other techniques that makes it more complex. Instead another approach is taken where the simplices on the boundary are locked. This means that the mesh operations are not allowed to change the boundary between two submeshes. Then both the red and black submeshes can be computed in parallel (except for the boundary). To compute the boundary, new submeshes must be created such that those simplices that were on the boundary before, are not on the boundary anymore. Even though the mesh operations are local, the effects may propagate to other parts of the mesh which means that it might be necessary to create new submeshes multiple times. This scheme will also be referred to as red-black ordering, where the first submeshes are part of the red step, and the black step is when new submeshes are created.

## 6.2 Static and Dynamic Task Assignment

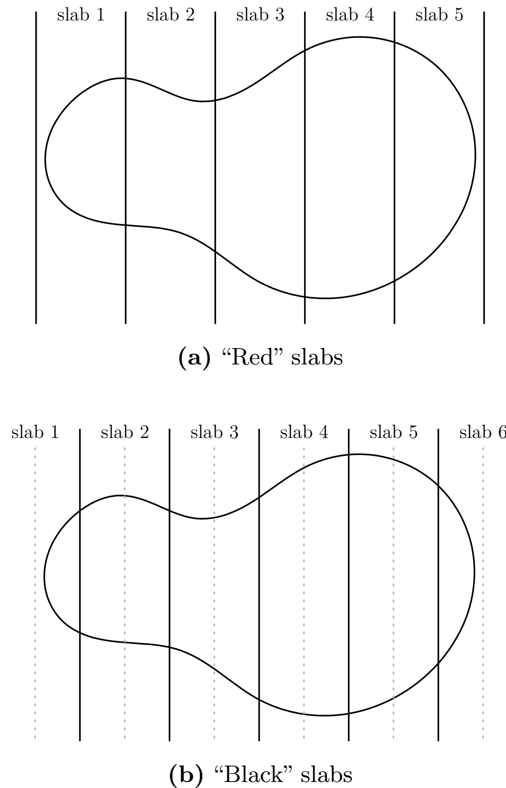
The computational work a process must do is called a task. There are generally two ways to assign tasks, static and dynamic task assignment [13]. Assume that we have exactly $n$ processors at our disposal. Static task assignment (STA) is when we divide the total amount of work into $n$ tasks and assign each processor with a task. When each processor is done, the entire work has been performed and the program can continue. With dynamic task assignment (DTA), the work is divided into more than $n$ tasks which are placed in a task queue. When a processor has finished a task, it will receive a new task from the queue of tasks, and this goes on until the queue is empty.

The advantage of STA is that it is simpler to implement the assignment of tasks because there is no queue to keep track of. On the other hand, for some problems it might be difficult to balance the tasks such that they take roughly the same time to compute and the entire process is determined by the processor which is slowest. The advantages of DTA is that you can have unbalanced tasks and still keep all processors busy because a processor that finished early will just get a new task assigned. It also supports systems where the number of processors that are available changes over time. Having small tasks might make the overhead of assigning tasks too high, while too large tasks might make it perform almost like STA because the last task still determines the total computational time. Some of the problems with both STA and DTA can be overcome with the work steal strategy. Work steal means that if one processor takes a very long time to finish a task, another task can take some of the workload, and the one stolen from, will finish faster. This complicates the

implementation and extra overhead may be created because there need to be some synchronization between processes. Static task assignment without any work steal is chosen because it is simpler and the benefits of work steal are not worthwhile given the time constraint of the project. Additionally, STA might be more suitable for an implementation on a GPU because all the processors are exactly the same.

## 6.3 Slabbing

One way to divide a mesh into submeshes is to imagine that we slice the mesh vertically (or horizontally) a number of times to create slabs, see Figure 6.2. Each simplex is assigned to the slab it is inside. These are the red regions, Figure 6.2a. The black ones are created by shifting the slabbing with the half width of a slab, Figure 6.2b. This makes the previously locked boundary unlocked and the computations can continue. The following is a discussion of two methods for slabbing, fixed-width method and counting method.




**(a)** “Red” slabs

**(b)** “Black” slabs

**Figure 6.2:** Examples of slabbing decomposition of mesh. Figure 6.2a shows the “red” slabbing and Figure 6.2b shows the “black” slabbing. The solid vertical lines show the slab boundaries and the dashed lines in Figure 6.2b show where the red slabs would be for comparison.

**Fixed-width method.** The simplest method is the fixed-width method where all the slabs have the same width. If the static task assignment is used and the

width of the entire mesh is $W$, the width $w$ of a slab would be $w = W/p$ where $p$ is the number of processors. If dynamic task assignment is used, the width would be $w = W/n$ where $n$ is integer and $n > p$. The advantages of this method is that the assignment of simplices can be done in parallel because it only depends on the coordinates of each simplex. However, the method may also make unbalanced tasks because some parts of the mesh might have a higher density of simplices than others.

**Counting method.** The counting method method is slightly more complicated than fixed-width method. Instead of fixing the width of a slab, the number of simplices in each slab is fixed. If the static task assignment is used and the number of simplices in the entire mesh is $S$, the number of simplices $s$ of a slab would be $s = S/p$ where $p$ is the number of processors. If dynamic task assignment is used, the number of simplices would be $s = S/n$ where $n$ is integer and $n > p$. The advantage of this method is that the slabs are balanced but the disadvantage is that the simplices have to be sorted which makes the slab assignment slower.

The procedure for creating slabs using the counting method can be seen in Algorithm 6.1 and the procedure for merging slabs can be seen in Algorithm 6.2. When the submeshes are recombined into one large mesh, the shared vertices (and edges) must be handled differently because otherwise they will be inserted twice (and the mesh will be disconnected). When a new submesh is created, the indices of the simplices will not remain the same because each submesh will have its own internal counter. This also means that two vertices which represent the same vertex in the original mesh are not likely to have the same index. This complicates the merging because the indices themselves do not tell us which other vertex from the neighbour slab is the other half of the pair that makes up the shared vertex. The vertices at the submesh boundary are not allowed to move, so one way to solve this problem is to find the vertex which is closest to the one we are looking at. That vertex will very likely be the one to merge with. The advantage of this method is that we do not have to know anything about the relationship between vertices in the original mesh, but the disadvantages are numerous: numerical problems, assumption of coordinates[4], time spent on searching for the closest vertex etc. The method chosen is therefore the following: Each submesh $i$ keeps a map $S'_i$ from the new submesh indices to the indices of the original mesh. This means that the indices of two vertices from different submeshes will map to the same index and they must therefore be treated as a single vertex when the submeshes are merged. This gives the extra memory used for the map, but it does not have the disadvantages of the previously explained method. This idea is visualized in Figure 6.3. Note that the slabbing method can be used for any mesh implementation (although it does require coordinates for sorting).

When the submeshes are created, the attribute vectors of the original mesh are invalidated because when the submeshes are merged, the indices will not match those of the original mesh. Therefore, when a submesh is created, new

[4]Splitting the meshes also assumes coordinates, but the merging procedure described can be used for other submeshing methods than slabbing which makes it desirable to have it as generic as possible.

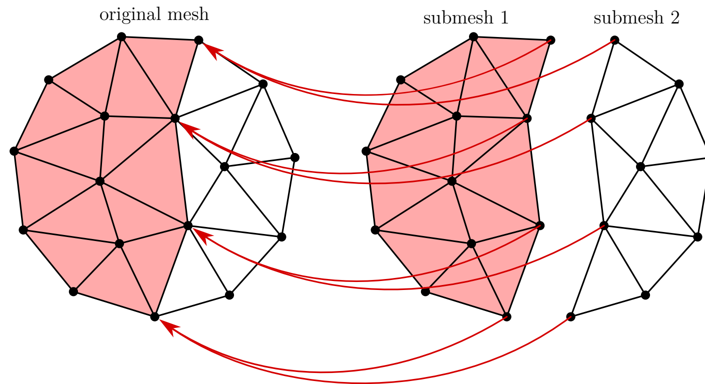



**Figure 6.3:** An example of how shared vertices are tracked. Two neighbouring submeshes, submesh 1 and submesh 2, each keep a map from vertices in their own mesh to the corresponding vertex in the original mesh. In this way, one can with certainty know which vertices are to be treated as one, because these vertices (and the edges between them) will not be changed.

attribute vectors are also created and the data is copied to these. When the submeshes are merged again, a new attribute vector is created, and the data from all the submeshes is copied into this one. The old attribute vector from the original mesh is deleted. This also solves the problem where the vertex indices grow during the simulation if the creation of new vertices does not reuse old indices (which is not the case for the mesh data structure chosen for this implementation). When the submeshes are merged, all vertices (and their data) which are unused, will not be copied to the new mesh. This can simplify an implementation because no procedures for reusing vertices are needed.

Even though the mesh operations are local, they do require some of the neighbouring simplices to be unlocked for the operations to work. For example, if a submesh consists of just a strip of triangles, using local retriangulation on any triangle will have no effect. This scenario can easily occur with the slabbing method when there are too few simplices (or too many slabs) or if the slabbings are vertical when the mesh more tall that wide. In some cases, the best slabbing slices might not be axis parallel but instead rotated at some arbitrary angle. In that case, the shape of the mesh has to be analyzed, e.g. by finding the minimal bounding box. Slabbing with the counting method is chosen because it is simple yet it does not suffer from balance issues with the fixed-width method. For irregular domains, like the example in Figure 6.1, the slabbing method might not be the best choice, but the implementation has support for changing the submeshing method so one can find another method that is better. In many cases, though, the domain is an axis-aligned rectangular shape because it just has to contain the interface.

## 6.4 Locked Simplices

During the simulations there are some simplices that are restricted in the sense that they cannot be moved or deleted (or both). Additionally, there are some

**Algorithm 6.1** Slabbing Procedure Create Submeshes

**Input:** Simplicial complex $K$ and number of submeshes $n$.

**procedure** CreateSubmeshes($K, n$)

  $N$ : 2-simplex $\mapsto \mathbb{Z}$ ▷ Mesh to submesh number

  $S_i$ : 0-simplex $\mapsto$ 0-simplex, $i \in \mathbb{Z}$ ▷ Submesh $i$ to mesh

  $n' \leftarrow$ **if** red **then** n, **otherwise** $n + 1$

  $\Sigma \leftarrow$ all 2-simplices of $K$

  $T \leftarrow$ sort $s \in \Sigma$ by Coord values

  $k \leftarrow \lfloor \frac{|\Sigma|}{n'} \rfloor$

  $r \leftarrow |\Sigma| \mod n'$

  Below, the $r$ first sets must each have 1 more element

  **if** red **then**

    $\Sigma'_i \leftarrow$ **closure**$(T[(i-1)k+1, ik])$ for $i \in \{1 \ldots n'\}$

  **else**

    $\Sigma'_1 \leftarrow$ **closure**$(T[1 \ldots \lfloor \frac{k}{2} \rfloor])$

    $\Sigma'_i \leftarrow$ **closure**$(T[\lfloor \frac{k}{2} \rfloor + (i-1)k + 1, \lfloor \frac{k}{2} \rfloor + ik])$ for $i \in \{2 \ldots n'\}$

  **end if**

  $N \leftarrow$ map from 2-simplices to submesh number $[1 \ldots n']$

  $M$ : 0-simplex $\mapsto$ 0-simplex ▷ Original mesh to submesh

  **for** $i \leftarrow 1 \ldots n'$ **do**

    **for all** $s \in$ **filter**$_0(\Sigma'_i)$ **do** ▷ Insert vertices

      $s' \leftarrow$ Insert$(K'_i)$

      $M[s] \leftarrow s'$

      Copy data $s \in K$ to $s' \in K'_i$

      **if** $s$ is on a submesh boundary as determined by $N$ **then**

        IsSubmeshBoundary$(s') \leftarrow$ True

        $S_i[s'] \leftarrow s$

      **end if**

    **end for**

    **for all** $s \in$ **filter**$_2(\Sigma'_i)$ **do** ▷ Insert triangles (and edges)

      Rename vertices in **closure**$(s)$ to $v_1$, $v_2$ and $v_3$

      $s' \leftarrow$ Insert$(\langle M[v_1], M[v_2], M[v_3] \rangle, K'_i)$

      Copy data $s \in K$ to $s' \in K'_i$ (including edges of $s$)

      Phase$(s') \leftarrow$ Phase$(s)$

    **end for**

  **end for**

  $n', S_i, K'_i$ for $i \in \{1 \ldots n'\}$ are saved for use in MergeSubmeshes

**end procedure**

**Algorithm 6.2** Slabbing Procedure Merge Submeshes

```
procedure MergeSubmeshes
    n', S_i, K'_i for i ∈ {1 ... n'} are created by CreateSubmeshes

    O : 0-simplex ↦ 0-simplex                          ▷ Original mesh to new mesh
    K' ← new simplicial complex

    for i ← 1 ... n' do
        S' : 0-simplex ↦ 0-simplex                     ▷ Submesh to new mesh
        Σ ← all simplices of K_i

        for all s ∈ filter_0(Σ_i) do                   ▷ Insert vertices
            s' ← S_i[s]

            if s' ≠ Null then     ▷ Does the vertex exist in the original mesh?
                j ← minimum of the two submesh numbers that s divides
                if i = j then                          ▷ Only insert the vertex once
                    s'' ← Insert(K')
                    O[s'] = s''
                    Copy data s ∈ K'_i to s'' ∈ K'
                end if
            else
                s'' ← Insert(K')
                S'[s] ← s''
                Copy data s ∈ K'_i to s'' ∈ K'
            end if
        end for
    end for

    for all s ∈ filter_2(Σ_i) do                       ▷ Insert triangles (and edges)
        Rename vertices in closure(s) to v_1, v_2 and v_3

        v'_j ← if S_i[v_1] ≠ Null then O[S_i[v_j]] otherwise S'[v_j]
            for j = {1, 2, 3}

        s' ← Insert(⟨v'_1, v'_2, v'_3⟩, K')
        Copy data s ∈ K'_i to s' ∈ K' (including edges of s)
        Phase(s') ← Phase(s)
    end for

    K ← K'
end procedure
```

restrictions to when new vertices can be inserted. A simplex that is restricted in any way is referred to as locked. Consider the mesh in Figure 6.4 which is rectangular and has two phases. The mesh is decomposed into four submeshes (slabs) represented by the dashed lines, and consider now the last submesh in red colour (the discussion applies to the other slabs as well). There are four types of locked vertices illustrated by red or black dots. First, consider the red mesh corner vertices marked with ①. If any of these vertices are moved or deleted, the shape of the domain will change, and this is not allowed and so these vertices must be locked during the entire simulation. The black vertices on the mesh boundary and the edges that connect them, marked with ②, can be deleted or moved but only such that the vertical or horizontal lines of the domain stay intact. Vertices can also be inserted under the same restrictions. This is true at all times during the simulation. The black vertices marked with ③ and the edges that are between them are locked because they are shared with the neighbouring submesh. When the submeshing changes colour (for slabbing, the slabs are shifted a half slab width), these will not be locked anymore and thus the lock status of these simplices changes during the simulation. Finally, the red vertices, marked with ④, are the vertices on the interface. These may only be moved as determined by the application and vertices (and edges) can only be inserted or deleted for either coarsening or refinement purposes.

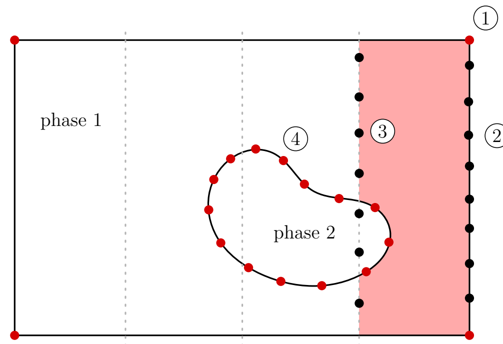
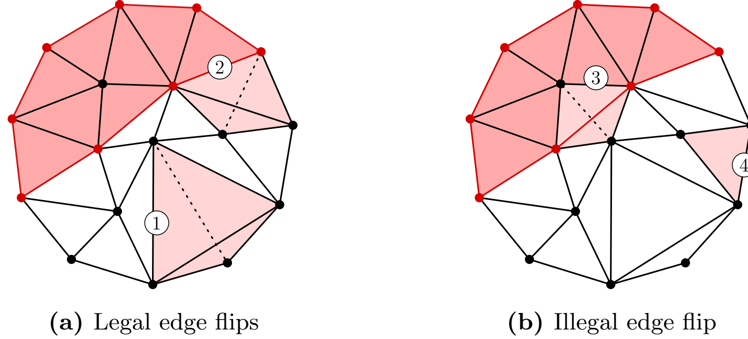



**Figure 6.4:** A mesh with two phases split into four slabs. The fourth red slab contains four types of locked vertices. Red vertices at ① cannot be moved or deleted because it will change the shape of the domain. Black vertices at ② can only be changed such that the shape of the domain is maintained. Black vertices at ③ cannot be changed at this time. Red vertices at ④ can only be moved as determined by the application results and can otherwise only be changed for coarsening or refinement purposes.

## 6.5 Mesh Operation Restrictions

As discussed in Section 6.4, mesh optimization algorithms are restricted with regard to what simplices they can be used on. These restrictions must be adhered to by the mesh operations introduced in Section 4.2. The following is an explanation of how this affects the mesh operations. It is assumed that the interface must be preserved, but note that in some cases, for specific applications, this may not always be the case.

**Edge flip.** If the two adjacent 2-simplices (triangles) are in the same phase, edge flip is a legal operation (even when one of the triangles has an edge on the interface), see the examples marked with ① and ② in Figure 6.5a. If the edge if part of the interface between two phases, edge flip is *not* legal because it would change the interface, see the example marked with ③ in Figure 6.5b. If the edge is on the mesh boundary, see ④, or on the boundary between two submeshes, edge flip is also not legal.




**(a)** Legal edge flips

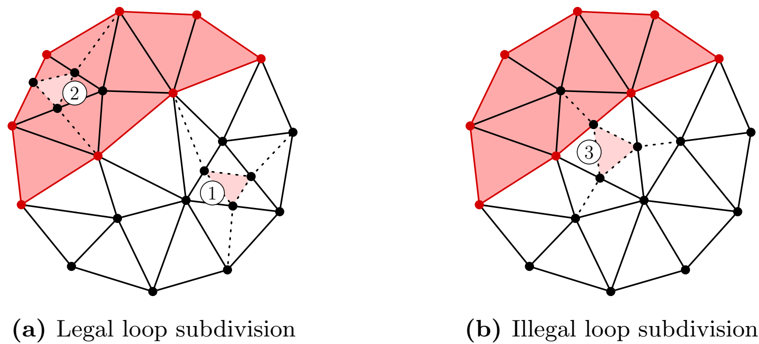
**(b)** Illegal edge flip

**Figure 6.5:** Examples of legal and illegal uses of edge flip. Figure 6.5a and Figure 6.5b are meshes with a red phase and a white phase, divided by a red interface. The dashed lines are new edges after an edge flip within the light red triangles. ① and ② are legal edge flip operations because operation does not change the interface. ③ is illegal because the operation changes the interface. ④ is illegal because it is on the mesh boundary.

**Loop subdivision.** When the triangle to be subdivided does not have an interface edge, the operation is legal because it does not change the interface, see the example marked with ① in Figure 6.6a. The example marked with ② is also legal because the mesh boundary is allowed to be refined. Note that in this case, one less triangle is split. On the other hand, if the triangle has an interface edge, a new vertex is created on the interface and an edge is added to the adjacent phase, see ③ in Figure 6.6b. This is considered illegal because it changes the quality of the interface.[5] If the triangle is on a boundary between two submeshes, we cannot split all three adjacent triangles, because that will change the neighbouring submesh, but we also cannot omit to split that triangle like on the mesh boundary, because that will create a mesh which is inconsistent and not a simplicial complex, see Figure 6.7. In this case, the problem may disappear after remeshing because the triangle is not longer on a submesh boundary and loop subdivision can therefore be used.

**Local retriangulation.** The triangles that are affected using local retriangulation are the input triangle and the ones that share a vertex with the input triangle. The number is varying and it is therefore not possible to enumerate all possibilities as it is with for example loop subdivision where at most four triangles are affected. If the triangle chosen is does not have an interface vertex, then

[5]In some cases it might be considered legal to change the interface when using loop subdivision, but because the interface has its own quality measure, this approach has been chosen to maintain control of the interface.




**(a)** Legal loop subdivision **(b)** Illegal loop subdivision

**Figure 6.6:** Examples of legal and illegal uses of loop subdivision. Figure 6.6a and Figure 6.6b are meshes with a red phase and a white phase, divided by a red interface. The dashed lines are new edges added after loop subdivision with the light red triangles as input. ① is legal because the interface is not changed. ② is legal because the mesh boundary is refined, but not changed. ③ is illegal because the interface is changed.

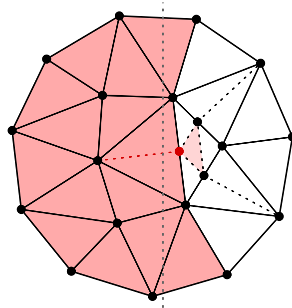
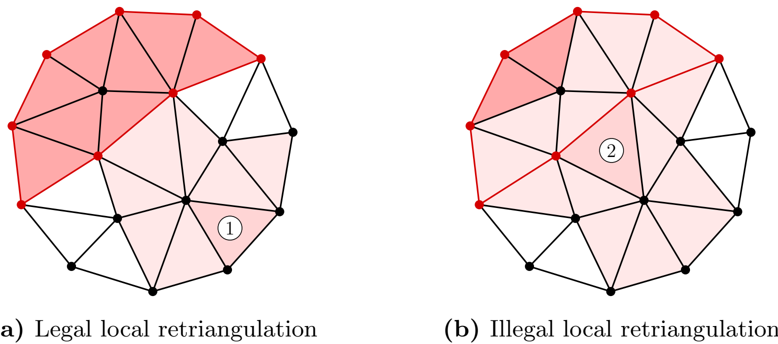


**Figure 6.7:** Example of an illegal use of loop subdivision. A subset of a mesh is divided into two submeshes, a red one and a white one, by a vertical dashed line. The light red triangle and dashed edges are inserted by loop subdivision. The red edge cannot be inserted because the triangle it divides does not belong to the white submesh and the red vertex cannot be inserted because it is on the submesh boundary. This operation is illegal.

the interface is unchanged, see Figure 6.8a. If a triangle *does* have an interface vertex, see Figure 6.8b, the interface would be removed and the new triangles may not restore the old interface. If the retriangulation algorithm supports fixed edges (that is, edges that must be present after the retriangulation), then the interface could be set to fixed, but that would still make changes to an adjacent phase which is not desirable because they can have different quality measures. If the triangle has vertices on a submesh boundary, the operation is also illegal because it cannot remove triangles in another submesh.

**Edge Split.** If the edge is not an interface edge and it is a subface of two triangles, edge split is legal, see the example marked as ① in Figure 6.9. Because edge split is used on edges, it is legal to use it on an interface edge to refine it, see ②. When used on a mesh boundary, it only splits one triangle, see ③, but this is legal because it refines the mesh boundary. If the edge is on a submesh boundary, see the example in Figure 6.10, the operation is illegal because it




**(a)** Legal local retriangulation

**(b)** Illegal local retriangulation

**Figure 6.8:** Examples of legal and illegal uses of local retriangulation. Figure 6.8a and Figure 6.8b are meshes with a red phase and a white phase, divided by a red interface. If the triangle at ① is chosen as input, the interface is not changed and the operation is legal. If ② is chosen, the interface is changed, and the operation is illegal.

changes triangles in a neighbour submesh.

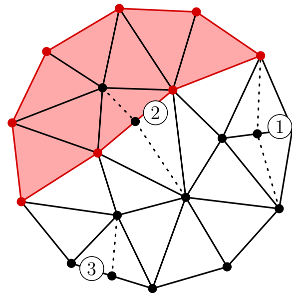



**Figure 6.9:** Examples of legal uses of edge split. A mesh with a red phase and a white phases, divided by a red interface. ① is legal because all edges are of the same submesh. ② is legal when refining the interface. ③ is legal if the edge where the vertex is inserted is on the mesh boundary because it is legal to refine the mesh boundary as long as it does not change the shape of it.

**Edge collapse.** If the edge has no interface vertices or no boundary vertices of any type, it is legal to perform edge collapse by collapsing to either end of the edge, see ① in Figure 6.11a. Because edge collapse is used on edges, it is legal to use it on an interface edge to coarsen it, see ②. If the edge has exactly one vertex on a boundary, see ③, only the vertex which is not on the boundary can be moved. In Figure 6.11b, ④ is not legal because it changes the mesh boundary (note that it is the opposite direction as ③ in Figure 6.11a). ⑤ is also not legal because it also changes the interface. In the case, however, where the domain is for instance a rectangle and the edge does not touch any of the corners, edge collapse would not change the shape of the rectangle and it would be legal to perform the operation. At last, ⑥ is illegal because it changes the interface.[6]

[6]In some cases it might be considered legal to collapse an edge between an interface vertex and a non-interface vertex. This depends on the application.

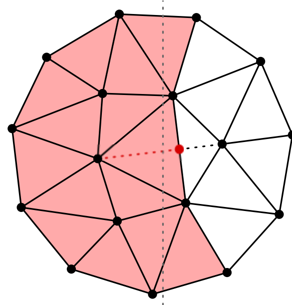
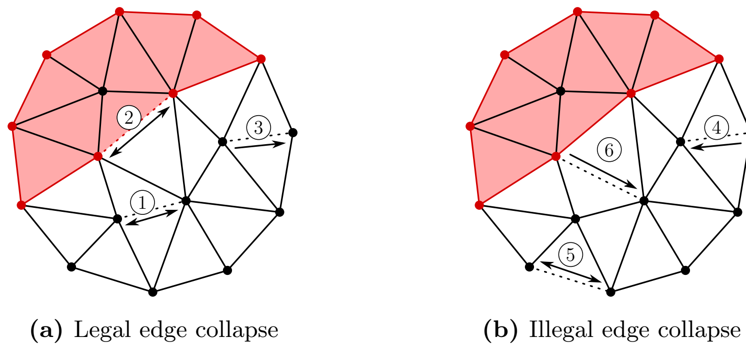


**Figure 6.10:** Example of an illegal use of edge split. A subset of a mesh is divided into two submeshes, a red one and a white one, by a vertical dashed line. The dashed edges are inserted by edge split. The red edge cannot be inserted because it does not belong to the white submesh and the red vertex cannot be inserted because it is on the submesh boundary. This operation is illegal.




**(a)** Legal edge collapse **(b)** Illegal edge collapse

**Figure 6.11:** Examples of legal and illegal uses of edge collapse. Figure 6.11a and Figure 6.11b are meshes with a red phase and a white phase, divided by a red interface. An arrow next to an edge indicates which vertex is moved and merged with the other vertex of the edge. Double arrows indicates that both vertices are moved in the specific example. If the edge at ① is chosen, either end points can legally be moved. The same is true for ② when coarsening the interface. ③ can, on the other hand, only be moved one way because, because otherwise, as seen in ④ (which is illegal), it will change the mesh boundary. ⑤ is also illegal because it changes the mesh boundary and ⑥ is illegal because it changes the interface.

## 6.6 Parallel Algorithm

Using either the slabbing method in Section 6.3 or any other submeshing method that implements the methods CreateSubmeshes and MergeSubmeshes, the DSC algorithm in Algorithm 5.1 is run in parallel on each created submesh. As seen in Section 6.4, submesh boundary simplices cannot be changed by the DSC algorithm, and therefore CreateSubmeshes must be called again with a new submeshing (for example the slabbing method shifts the slabs by a half slab width). Because this creates changes to the mesh that might have influence on other parts of the mesh, e.g. an interface vertex that was previously stuck is suddenly free to move, the submeshes can be created and merged continuously until some stop criterion is met. An example of a simple stop criterion is "stop

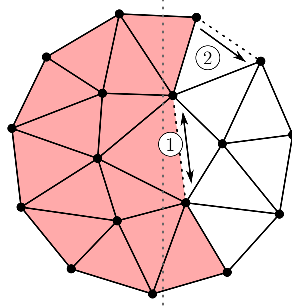



**Figure 6.12:** Example of an illegal use of edge collapse. A subset of a mesh is divided into two submeshes, a red one and a white one, by a vertical dashed line. An arrow next to an edge indicates which vertex is moved and merged with the other vertex of the edge. Double arrows indicates that both vertices are moved in the specific example. The dashed edges are used as input to edge collapse. No vertex can be moved in ① because no changes are allowed on the submesh boundary. At ②, the given movement is illegal because it will also change the submesh boundary.

when all interface vertices are moved to their target positions or if the number of iterations exceeds $x$", where $x$ is some positive integer. Note that the DSC has a similar criterion but when the mesh is divided into submeshes, it only has its given submesh available to compute the quality of, and therefore a quality criterion that is computed for the entire mesh is needed. The algorithm is shown in Algorithm 6.3.

---

**Algorithm 6.3** Parallel Deformable Simplicial Complex

---

**Input:** A simplicial complex $K$.

**procedure** ParallelDSC($K$)
  **while** $Q^K_{\text{mesh}}$ is too low **do**

    $K' \leftarrow$ CreateSubmeshes($K$)

    **for all** $K_i \in K'$ **do**
      Spawn process from DSC($K_i$)
    **end for**

    Join with all processes

    $K \leftarrow$ MergeSubmeshes($K'$)

  **end while**
**end procedure**

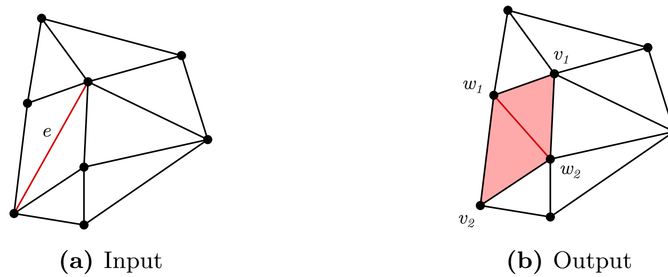
---

# 7 Mesh Algorithms

Mesh operation algorithms are expressed using simplex operations and relations because that makes them unaware of the underlying mesh data structure and they can be reused even though the data structure changes. To modify the simplicial complex $K$, mesh functions, as defined in Section 4.1, are used. The following are algorithms for the operations described in Section 4.2 with the necessary precautions taken to make sure that restrictions on the interface, mesh boundary and submesh boundary are respected. If a mesh operation cannot be used, it returns with the error “illegal operation”.

## 7.1 Edge Flip

Edge flip first checks if the input edge $e$ is on the interface, on the submesh boundary or the mesh boundary, see Algorithm 7.1. If this is the case, the operation returns with an error. Let $e$ be composed of the two vertices $v_1$ and $v_2$, see Figure 7.1, then two opposite vertices, $w_1$ and $w_2$, are found, and they will be the end points of a new edge. The two triangles that have $e$ as a subface must be removed and two new triangles are inserted.




**(a)** Input **(b)** Output

**Figure 7.1:** Illustration of edge flip algorithm. The red 1-simplex $e$ in Figure 7.1a is the input. The red simplices in Figure 7.1b are the new simplices. Note that $e$ and it shared triangles are removed. The inserted triangles are $\langle w_1, w_2, v_1\rangle$ and $\langle w_1, w_2, v_2\rangle$ and their edges. No new 0-simplices are inserted.

## 7.2 Loop Subdivion

The mesh operation loop subdivision subdivides a triangle $t = \langle A, B, C\rangle$ by inserting a smaller triangle inside it. The algorithm for loop subdivision is shown in Algorithm 7.4 and an illustration is shown in Figure 7.2. First the edges and the vertices of the triangle are stored in two separate sets. If any of the edges are either on the submesh boundary or on the interface, the operation is illegal and the operation halts. The (up to) three triangles that shares an edge with $t$ are found, and the vertices which are not shared by $t$, are named $A'$, $B'$ and $C'$. These are found using the helper function OppositeVertex in Algorithm 7.3. The new vertices $a, b$ and $c$, which are the midpoints of the edges of $t$, are found using the helper function ComputeMidpoint. The helper function TriangleSplit in Algorithm 7.2 splits the incident triangles. At last, all the remaining triangles are inserted. Note that if we are on the mesh boundary, $A'$, $B'$ or $C'$ might not exist, and therefore ignored in that case.

**Algorithm 7.1** Mesh Operation Edge Flip

**Input:** An edge $e \in K$.

**procedure** EdgeFlip($e, K$)
  **if** IsBoundary($e$) or
   IsSubmeshBoundary($e$) or
   IsInterface($e$) **then**
    **return** illegal operation
  **end if**

  $V \leftarrow$ **boundary**($e$)
  $T \leftarrow$ **star**($e$)
  $W \leftarrow$ **filter**$_0$(**closure**($T$)) $\setminus V$
  $p \leftarrow$ Phase($T_i$), $T_i$ is any simplex in $T$

  Rename 0-simplices in $V$ to $v_1$ and $v_2$
  Rename 0-simplices in $W$ to $w_1$ and $w_2$

  Remove($e$)
  Remove($T$)

  $t_1 \leftarrow$ Insert($\langle w_1, w_2, v_1 \rangle, K$)
  $t_2 \leftarrow$ Insert($\langle w_1, w_2, v_2 \rangle, K$)

  Phase($t_1$) $\leftarrow p$
  Phase($t_2$) $\leftarrow p$
**end procedure**

**Algorithm 7.2** Helper Operation Triangle Split

**Input:** A triangle $t \in K$, a vertex $v_0 \in$ **boundary**($t$) to split from and a vertex $m_0$ on the midpoint of the opposite edge.

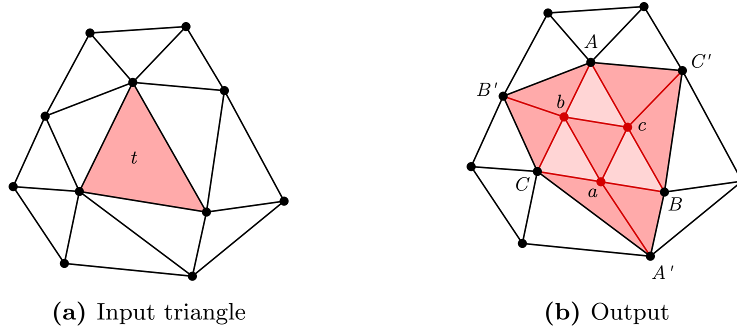
**procedure** TriangleSplit($t, v_0, m, K$)
  $p \leftarrow$ Phase($T$)
  $V \leftarrow$ **filter**$_0$(**boundary***($t$)) $\setminus \{v_0\}$

  Remove($t, K$)

  Rename vertices in $V$ to $v_1$ and $v_2$

  Insert($\langle m, v_0, v_1 \rangle, K$)
  Insert($\langle m, v_0, v_2 \rangle, K$)

  Phase($\langle m, v_0, v_1 \rangle$) $\leftarrow p$
  Phase($\langle m, v_0, v_2 \rangle$) $\leftarrow p$
**end procedure**




**(a)** Input triangle

**(b)** Output

**Figure 7.2:** Illustration of loop subdivision algorithm. The red 2-simplex $t$ in Figure 7.2a is the input. The red simplices in Figure 7.2b are the new simplices. Note that $t$, the edges of $t$ and the three triangles that shares an edge with $t$ are removed. The inserted triangles, including their subfaces, are $\langle a, b, c\rangle$, $\langle A, b, B'\rangle$, $\langle B', b, C\rangle$ etc.

**Algorithm 7.3** Helper Operation Opposite Vertex

**procedure** OPPOSITEVERTEX($t, e$)
  $T \leftarrow$ **star**($e$)

  **if** $|T| = 1$ **then**
    **return** nil
  **end if**

  $T \leftarrow T \setminus \{t\}$
  $V \leftarrow$ **closure**($T$) $\setminus$ **boundary**($e$)

  Rename vertex in $V$ to $v$

  **return** $v$
**end procedure**

## 7.3 Local Retriangulation

The polygon to retriangulate is found by selecting a subset of simplices and choosing the boundary of the subset as the polygon. The subset has to have at least one vertex which is not on the polygon, because otherwise the new triangulation will have the same number of triangles. If too many simplices are chosen, the result might be that the mesh is oversimplified and too many details are removed. There are multiple ways to select the simplices, but it seems reasonable to choose simplices that are as close to each other as possible. The input is a triangle $t$ and the chosen simplices are $R =$ **star**(**closure**($T$)), the immediate neighbours.[7] Using the helper functions PHASEFILTER, BOUNDARYFILTER and SUBMESHBOUNDARYFILTER, we remove the triangles from other phases than $t$ as well as any submesh boundary or mesh boundary simplices (vertices and edges) there might be because we cannot remove these. The polygon to retriangulate is then $R' =$ **closure**($R$) $\setminus R$. If $R'$ contains two or more polygons

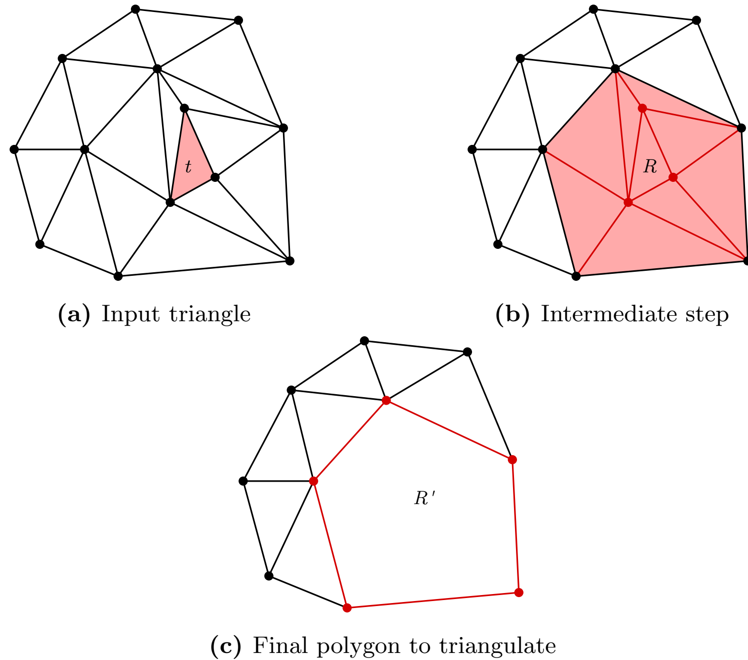
[7] One can experiment with using closure and star multiple times to increase the number of simplices chosen.

**Algorithm 7.4** Mesh Operation Loop Subdivision

**Input:** A triangle $t \in K$.

```
procedure LoopSubdivision(t, K)
    E ← boundary(t)
    V ← filter_0(boundary*(t))
    p ← Phase(t)

    if IsSubmeshBoundary(E) or IsInterface(E) then
        return illegal operation
    end if

    Rename vertices in V to A, B and C

    a ← Insert(K)
    b ← Insert(K)
    c ← Insert(K)

    Coord(a) ← ComputeMidpoint(B, C)
    Coord(b) ← ComputeMidpoint(C, A)
    Coord(c) ← ComputeMidpoint(A, B)

    if v ← OppositeVertex(⟨B, C⟩, t) ≠ nil then
        TriangleSplit(⟨A′, B, C⟩, v, a, K)
    end if

    if v ← OppositeVertex(⟨A, C⟩, t) ≠ nil then
        TriangleSplit(⟨A, B′, C⟩, v, b, K)
    end if

    if v ← OppositeVertex(⟨A, B⟩, t) ≠ nil then
        TriangleSplit(⟨A, B, C′⟩, v, c, K)
    end if

    Remove(t)
    Remove(E)

    t_1 ← Insert(⟨a, b, c⟩, K)
    t_2 ← Insert(⟨A, b, c⟩, K)
    t_3 ← Insert(⟨a, B, c⟩, K)
    t_4 ← Insert(⟨a, b, C⟩, K)

    Phase(t_i) ← p, i ∈ {1, 2, 3, 4}
end procedure
```

(because some simplices were removed, creating a hole that divides the simplices into two groups) or when the polygon is not simple, the operation returns with an error because not all triangulation methods support these. Otherwise, we can now remove $R$ and run the ear clipper on $R'$. The algorithm is shown in Algorithm 7.5 and the illustration of finding $R'$ is shown in Figure 7.3.




**(a)** Input triangle

**(b)** Intermediate step

**(c)** Final polygon to triangulate

**Figure 7.3:** Illustration of preparations for local retriangulation algorithm. The red 2-simplex $t$ in Figure 7.3a is the input. The chosen simplex set $R = \textsf{star}(\textsf{closure}(T))$ contains the red simplices in Figure 7.3b. $R$ is removed and the polygon to triangulate are the red simplices of the simplex set $R' = \textsf{closure}(R) \setminus R$ in Figure 7.3c.

To retriangulate the polygon $R'$ in Figure 7.3c, the ear clipping method is chosen because it is simple to implement and it gives reasonable results. It is based on the implementation in [10] and only supports simple polygons. It works in the following way. Assume that the $n$ vertices of the polygon $R'$ are given in counter-clockwise order in a linked list where we can remove objects in constant time. Now consider three consecutive vertices $v_{i-1}$, $v_i$ and $v_{v+1}$. Standing in vertex $v_{i-1}$, if walking towards $v_{v+1}$ through $v_i$ makes a left turn, then $v_i$ is an ear and the triangle $\langle v_{i-1}, v_i, v_{i+1} \rangle$ is chosen in the final triangulation. When a triangle is chosen, the middle vertex associated with the ear is removed and the two neighbour vertices are rechecked for whether they are ears or not. The algorithms starts by determining which vertices are ears for the entire polygon. This can be done in $O(n)$ time because checking one ear takes constant time and there are $n$ vertices. After this is done, the main algorithm starts by finding an ear, choosing the triangle, removing the vertex associated with the ear and updating the status of the neighbours. Finding an ear takes $O(n)$ time, choosing the triangle and updating the neighbours takes $O(1)$ time so it takes $O(n)$ per ear, and in total we get $O(n^2)$ for the $n$ vertices because we remove one vertex (the ear) every time. The entire algorithms therefore runs in $O(n+n^2) = O(n^2)$

time.

An example of the ear clipper is shown in Figure 7.4. Figure 7.4a shows the state after initialization of ears. Red vertices are ears and the black vertices are not. The numbers at the vertices indicate the order the vertices are traversed. The vertex with number $i$ will be referred to as $v_i$. First, $v_1$ is considered (as indicated by the dashed circle), but since it is not an ear, $v_2$ is selected in Figure 7.4b. Vertex 2 is an ear and $\langle v_1, v_2, v_3 \rangle$ is cut off (indicated by the red colour). This is the first triangle in the triangulation. The ear status of $v_1$ and $v_3$ are updated, and $v_1$ becomes an ear. Moving on to $v_3$ in Figure 7.4c, an ear is cut of at $v_3$, but no vertices changes ear status. In Figure 7.4d, $v_4$ is an ear and $v_5$ changes status to an ear. In Figure 7.4e, $v_5$ is also an ear. In Figure 7.4f, $v_6$ is skipped because it is not an ear, but $v_7$ is, and it is cut off. The remaining ear is cut off in Figure 7.4g which is the final triangulation. Note that it does not necessarily produce a "good" triangulation. The edge $\langle v_1, v_6 \rangle$ could be flipped to create triangles that are more equilateral. Whether those are better triangles depends on the application of course.

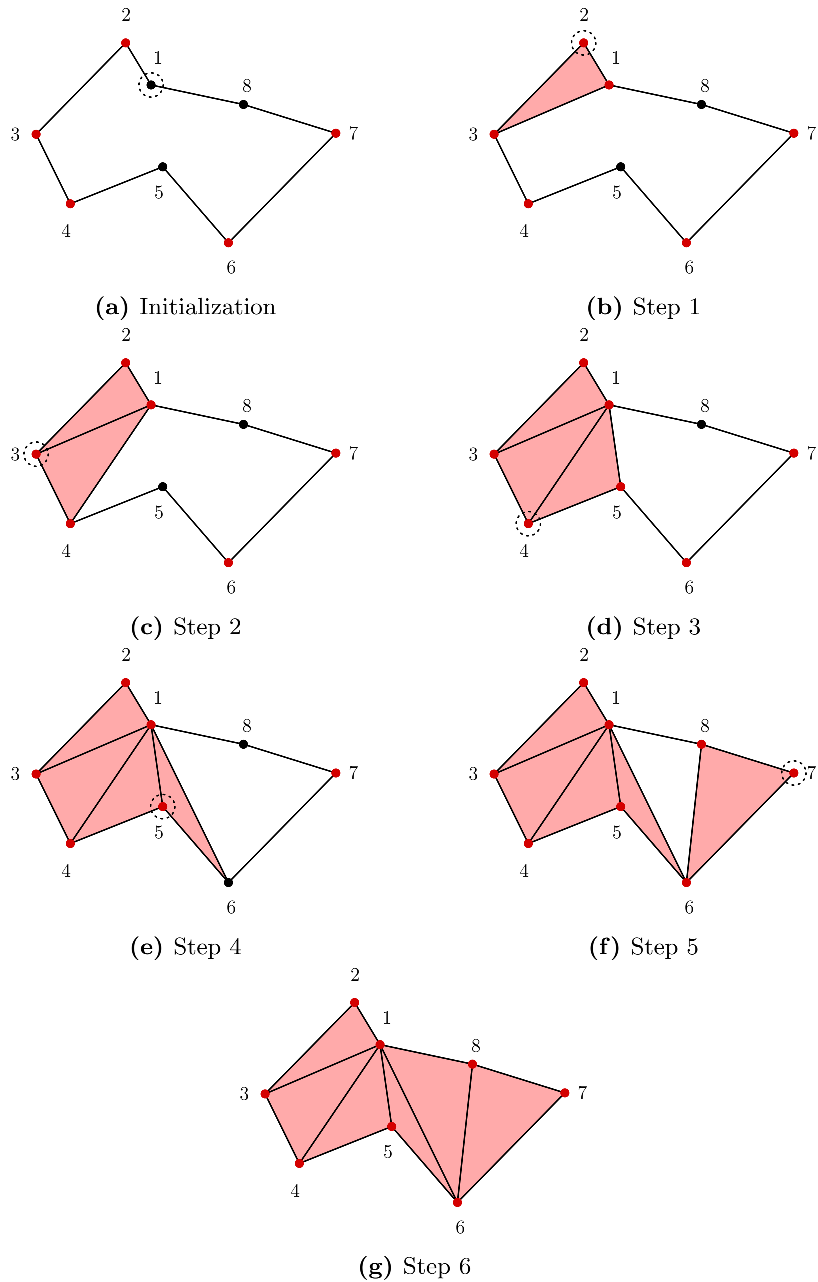
**Algorithm 7.5** Mesh Operation Local Retriangulation

**Input:** A triangle $t \in K$.

```
procedure LocalRetriangulation(t, K)
    R ← star(closure(t))
    PhaseFilter(t, R)
    BoundaryFilter(R)
    SubmeshBoundaryFilter(R)

    R' ← closure(R) \ R

    if R' is not a single simple polygon then
        return illegal operation
    end if

    p ← Phase(t)

    Remove(R, K)

    Retriangulate polygon with 0-simplices in R' as vertices

    Phase(t_i) ← p, for all 2-simplices t_i inserted in K
end procedure
```

## 7.4 Edge Split

Edge split uses the helper function ComputeMidpoint to find the midpoint between the two end points of the input edge $e$. The two triangles that share $e$ are split using the helper function TriangleSplit in Algorithm 7.2. This means that two or four triangles are created (depending on whether one or two triangles share $e$), see Algorithm 7.6 and Figure 7.5. The only edge which cannot be split is one on the submesh boundary (when that is the case, the operation



**(a)** Initialization

**(b)** Step 1

**(c)** Step 2

**(d)** Step 3

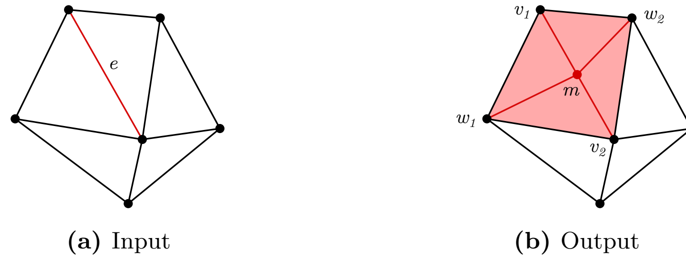
**(e)** Step 4

**(f)** Step 5

**(g)** Step 6

**Figure 7.4:** Example of ear clipping. Figure 7.4a shows input polygon after the ear initialization. Ears are marked with a red vertex and the rest are black. The numbers indicate the order the vertices are traversed. Figure 7.4b through Figure 7.4g each show a step when an ear is cut off. These are shown as red triangles. The vertex considered in each step is shown with a dashed circle. Note that vertices that are not ears are skipped. When an ear is cut off, the ear status of the two other vertices is updated.

returns with an error).

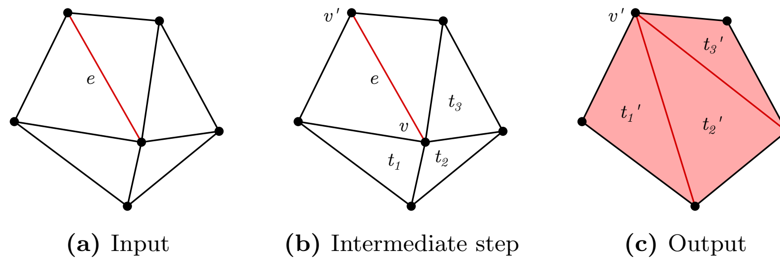



**(a)** Input **(b)** Output

**Figure 7.5:** Illustration of edge split. The red 1-simplex $e$ in Figure 7.5a is the input. The red simplices in Figure 7.5b are the new simplices. Note that $e$ and it shared triangles are removed. A new vertex $m$ is inserted halfway between $v_1$ and $v_2$. Four new triangles and their edges are inserted: $\langle v_1, w_1, m\rangle$, $\langle v_2, w_1, m\rangle$, $\langle v_1, w_2, m\rangle$ and $\langle v_2, w_2, m\rangle$.

## 7.5 Edge Collapse

An edge is collapsed into a single point with the edge collapse operation, see Figure 7.6 and Algorithm 7.7. One of the end points of the input edge $e$ is chosen as the point where the edge collapses to. In some situations this creates inverted triangles, see the example in Figure 7.7, which means that both end point are tried (unless the vertex is on the submesh boundary). If none of the end points create valid geometry, the operation returns with an error.




**(a)** Input **(b)** Intermediate step **(c)** Output

**Figure 7.6:** Illustration of edge collapse algorithm. The red 1-simplex $e$ in Figure 7.6a is the input. In Figure 7.6b is just before $v$ is removed and the triangles $t_1$, $t_2$ and $t_3$ are updated to use $v'$ instead and becomes $t'_1$, $t'_2$ and $t'_3$. The remaining triangles are removed. The result is shown in Figure 7.6c where the red simplices are the new simplices.

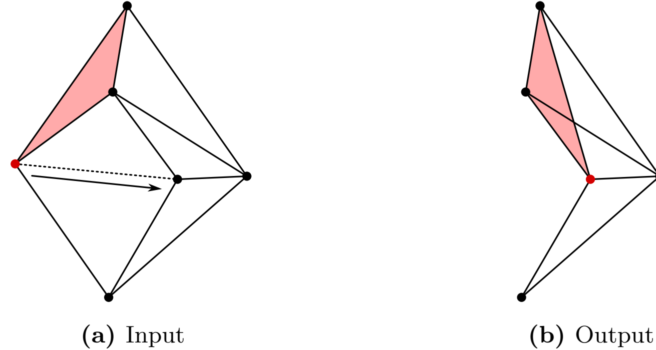
## 7.6 Summary

The simplex relations and operations are very suitable for navigating and querying the simplicial complex. The idea of using sets is mostly useful, but sometimes having a linear list with an ordering is necessary. For example in the local retriangulation, the order of the vertices of the polygon is needed for the ear clipper, but this is not expressible using sets. Additionally, when the polygon that must be retriangulated is determined, the actual retriangulation procedure

**Algorithm 7.6** Mesh Operation Edge Split

**Input:** An edge $e \in K$.

```
procedure EdgeSplit(e, K)
    if IsSubmeshBoundary(e) then
        return illegal operation
    end if

    V ← boundary(e)
    Rename vertices in V to v_1 and v_2

    m ← Insert(K)
    Coord(m) ← ComputeMidpoint(v_1, v_2)

    T ← star(e)

    Remove(e, K)

    for all t ∈ T do
        w ← 0-simplex in filter_0(boundary*(t)) \ V

        TriangleSplit(t, w, m, K)
    end for
end procedure
```



**(a)** Input **(b)** Output

**Figure 7.7:** Triangle inverted with edge collapse. Moving the red 0-simplex along the dashed edge in Figure 7.7a causes the red 2-simplex to invert as shown in Figure 7.7b. This is an illegal use of edge collapse.

is not written using this topology algebra (the implementation in [10], which it is inspired by, uses a typical pointer-based approach). The other mesh operations, however, are implemented using only the mesh functions and simplex relations and operations, so it might be necessary to replace the ear clipper with another method if a mesh is to be implemented elsewhere. The restrictions regarding the interface and boundaries are simple to implement as they often only require about two checks per operations and nothing else.

**Algorithm 7.7** Mesh Operation Edge Collapse

**Input:** An edge $e \in K$.

```
procedure EdgeCollapse(e, K)
    if IsSubmeshBoundary(e) then
        return illegal operation
    end if

    V ← boundary(e)
    T ← star(e)

    for all v ∈ V do
        if IsSubmeshBoundary(v) then
            continue
        end if

        V′ ← boundary(e) \ {v}
        T′ ← star(v) \ T

        Rename vertex in V′ to v′

        c ← Coord(v)
        Coord(v) ← Coord(v′)

        if any triangles in T′ are inverted then
            Coord(v) ← c
            continue
        end if

        for all t ∈ T′ do
            p ← Phase(t)
            t′ ← t, v substituted with v′

            Remove(t, K)
            Insert(t′, K)
            Phase(t′) = p
        end for

        Remove(v, K)
return
    end for
     return illegal operation
end procedure
```

# 8 Implementaton

The following is a description of the implementation specific details.

## 8.1 Cross-Platform Support

The programming language chosen for the implementation is C++ because it is supported on many platforms and it is a popular language. As concluded in [8], even though the programming language is supported, there were problems with compatibility in the 3D version of DSC because it was developed using a single platform for an extended period of time, and that meant that the project files for other platforms were outdated. This was fixed by using CMake which is a cross-platform build system where configuration files are created independently of platform with details about source files and dependencies. When a developer needs to work on the project, platform specific project files will be created by CMake. These project files should not be shared with other members of the development team or added to the repository. This project has been setup using CMake from the beginning with a clear separation of library files and application files.

## 8.2 Simplex Programming Interface

A new data structure implementation has to adhere to the simplex programming interface (API) such that the deformable simplicial complex algorithm can use it. In practice, this means that a new implementation must inherit from a mesh API class and override the virtual methods. A simplex $\sigma$ is represented by its vertices, $\mathsf{vert}(\sigma)$, which means that a 0-simplex (vertex) is represented by a single vertex, a 1-simplex is represented by two vertices and a 2-simplex is represented by three vertices. A vertex is represented by a unique number. Note that the data structure might internally use a different system, but a wrapper class can be created which has an instance of the data structure as a private member. A simplex set can contain simplices of any dimension, but in the implementation they are grouped by dimension because it makes iterating simpler (and more efficient because there is no need to test a simplex for dimension). This means that a simplex set contains three sets: one with vertices, one with edges and one with triangles. Each set is implemented with `std::set` from the C++ standard library because it removes duplicate entries automatically. The following is examples of the API that a simplicial complex mesh must implement, using the syntax

$$\text{return type } \textbf{methodName}(\text{argument list})$$

for simplex relations and operations.

**Simplex relations.** A simplex relation can be used on any simplex of arbitrary dimension. The three methods that must be implemented for the boundary relation are

$$\Sigma\ \textbf{boundary}(\sigma_0),\quad \Sigma\ \textbf{boundary}(\sigma_1)\quad \text{and}\quad \Sigma\ \textbf{boundary}(\sigma_2).$$

The other simplex relations have similar definitions. The semantics of the simplex relations can be found in Section 3.1

**Simplex operations.** A simplex operation can be used on any simplex of arbitrary dimension, but unlike simplex relations, they can also be used on a simplex set. The four methods that must be implemented for star are

$$\Sigma\ \mathsf{star}(\sigma_0),\quad \Sigma\ \mathsf{star}(\sigma_1),\quad \Sigma\ \mathsf{star}(\sigma_2)\quad \text{and}\quad \Sigma\ \mathsf{star}(\Sigma).$$

The other simplex operations have similar definitions. The semantics of the simplex operations can be found in Section 3.1

**Other methods.** For the implementation of insertion and removing simplices, the handling of edges has not been necessary because the computational domains are manifolds which means that edges do not exist without a the existence of a superface (triangle). Insertion of edges will therefore always be paired with an insertion of a triangle (if the edge does not already exist). Insertion of triangles requires no specific order of the vertices. When the mesh requires this (which is the case for the chosen mesh in this thesis, see Section 8.4), the implementation must be able to handle all orientations and change the ordering to fit the internal mesh data structure. Note also that it is the responsibility of the mesh to store information about boundaries that is given by the submeshing method. This is done to ensure that a mesh is not dependent on information that might not be available in distributed memory settings.

## 8.3 Attribute Vectors

An attribute vector is a map from a simplex (key) to a value. The type of the values is parametrized using C++ templates which means that any type can be used for the values, e.g. vector, real, integer etc. The 0-simplex attribute vector is implemented as a `std::vector` where the id of the simplex is used as input to the vector. The 1-simplex and 2-simplex attribute vectors are implemented using `std::map` where the simplices are used as keys in the map.

## 8.4 Mesh Data Structure

The chosen data structure for the mesh is PolyMesh from OpenTissue[8]. PolyMesh is a half-edge data structure which means that all edges are decomposed into two half-edges with opposite orientations. Insertion of new polygons (triangles in this case), must always be done with the same orientation (clockwise or counter-clockwise) because that is a constraint of the half edge data structures. The mesh contains coordinate information per vertex, but the coordinates will instead be saved in an external coordinate vector, and therefore all vertices will be inserted in $(0, 0)$. This has the consequence that upon insertion of triangles, the coordinates must be known such that the orientation can be calculated. This is problematic because DSC itself is independent of coordinates. One way to get around this would be to see which half-edge exists when appending a triangle to another, but that would mean that triangles could not be inserted in random order. Instead, this specific mesh implementation must be instantiated with a reference to the coordinates (passed to the constructor), while other implementations might not have this requirement.

[8]OpenTissue website: `http://www.opentissue.org`

## 8.5 Slabbing

The slabbing method implemented is using the counting method by implementation of Algorithm 6.1 and Algorithm 6.2, although instead of creating two submeshes of half the size of the others every other time, the first and last submeshes are combined into one. This ensures that the number of submeshes is constant (and not alternating between $n$ and $n+1$ where $n$ is the input to CreateSubmeshes). This is better when parallellising because it gives better control of the number of processes that must be created. An example of result of the slabbing implementation is shown in Figure 8.1.

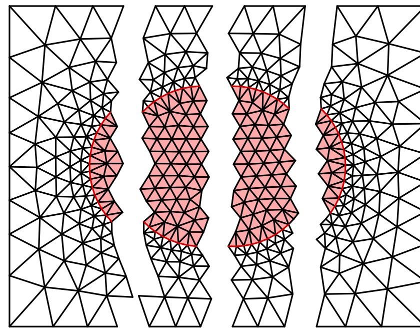


**Figure 8.1:** An example of the slabbing method for creating submeshes of a mesh. Vertices are omitted for readability. The number of triangles in each submesh is 174, 174, 173 and 173.

## 8.6 Interface and Boundaries

An implementation must know whether a simplex is on the mesh boundary, submesh boundary, on the interface or not. The following is a description of what information the implementation stores and what information is calculated.

**Phase.** Each triangle can only be part of exactly one phase, and therefore an attribute vector is created to store a phase id which is a number that represents a phase. Each phase is therefore represented by a phase id. Note that the exterior also has a phase id.

**Submesh boundary.** When creating the submeshes, the submeshing method provides the information of which vertices that are on the submesh boundary. This information is given to the mesh which is now able to determine whether a vertex is on the submesh boundary or not. An edge is on the submesh boundary if the two vertices of its end points are on the submesh boundary, except in the case where the edge is shared between two triangles, see Figure 8.2.

**Mesh boundary.** An edge is not on the mesh boundary if it is shared between exactly two triangles. If the edge is only a subface of one triangle, then the edge is on the mesh boundary if it is not on the submesh boundary. A vertex is on the mesh boundary if any of the edges it is connected to is on the mesh boundary.

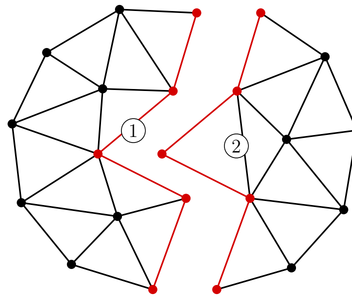



**Figure 8.2:** An example of submesh boundaries of a mesh with two submeshes. The simplices on the submesh boundary are shown in red. The edge at (1) is on the submesh boundary because its two end points are on the submesh boundary and only one triangle has the edge as subface. The edge at (2) is not on the submesh boundary because even though the two vertices are on the submesh boundary, the edge has two triangles as superface.

**Interface.** An edge is an interface if the two triangles that shares the edge is of different phases. If the edge is on a submesh boundary, we do not know the phase of one of the triangles, but because the submesh boundary is more restricted than the interface (as discussed in Section 6.4), changes to those edges and vertices are not allowed in any case. If the edge is on a mesh boundary, we check whether the phase (of the triangle that it is a subface of) has the phase id of the exterior. If not, then it is an interface. This makes sure that if the application is for example a simulation of water in a container with air, then when the water comes in contact with the container, then those edges will become interfaces, see Figure 8.3. A vertex is on the interface if any of the edges it is connected to is an interface. Note that the interface is represented *implicitly*.

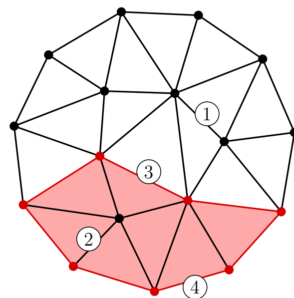



**Figure 8.3:** An example of a mesh with two phases. The phase of the exterior is shown in white and the other phase is shown in red. The interface (vertices and edges) is shown in red. The egdes at (1) and (2) are not part of an interface because the triangles sharing them are of the same phase. The edge at (3) is part of the interface because the triangles shared by it are of different phases. The edge at (4) is also part of the interface because the phase of the triangle is not the phase of the exterior.

## 8.7 Mesh Operations

The mesh operations for changing the mesh are implemented as described in Section 7 with the simplex programming interface described in Section 8.2.

## 8.8 Mesh Quality

The quality of a simplex is defined by a simplex quality measure as discussed in Section 4.3. A method for choosing which simplices to change is defined by a quality comparer which must, using the quality measure given, select the simplices that must be changed. A possible strategy is to compute the quality of all simplices and then sort them and then select the $n$ worst where $n$ is some integer number. Another strategy would be to select the simplices with a quality below some number. In this implementation, because all strategies must follow the same programming interface, they can be given any quality measure and choose the simplices to update according to their specific selection strategy. This also mean that the strategy can be changed and reused to match the requirements of the application.

## 8.9 Interpolators

When new simplices are inserted in the mesh, interpolator functions are called. These can be specified by the application to meet the requirements for the application. For example, if a vertex is inserted on the interface, an interpolator could compute a new value by taking the average of the neighbour vertices.

## 8.10 Sequential DSC and Parallel DSC

Two implementations are created for running a DSC algorithm on a mesh, a sequential version and a parallel version. Both are given a DSC algorithm behaviour which can be changed if the default (see Algorithm 5.1) does not meet the requirements of the application. Sequential DSC runs the given DSC algorithm every time there is a time step. It is mostly used for debugging and for comparison (time and mesh quality) with parallel DSC. The parallel version of DSC must, in addition to a DSC algorithm behaviour, be given a submeshing method. When there is a time step, parallel DSC uses the submeshing method to create new submeshes and then it runs the given DSC algorithm on all submeshes in parallel. When all are done, the submeshes are merged and if the stop criterion has been met, it returns, otherwise it repeats. In this way, new applications can more easily be created because a default behaviour is given as a starting point. When the more experience is gathered, a new algorithm can be created using the existing quality measures, mesh operations etc., or new ones can be created. Parallellism, however, does not need to be reimplemented because it works for any DSC algorithm behaviour.

## 8.11 Visualization

The mesh is exported to the XML-based format Scalable Vector Graphics (SVG). These files can be viewed in a modern web browser or in Inkscape.[9] This has

[9] Inkscape website: `http://www.inkscape.org`

the advantage that is does not create extra dependencies because the text file can be written using file streams from the standard library. The 3D version of DSC did not only have dependencies for vector libraries etc., but it also relied on multiple graphics libraries (which are sometimes less cross-platform) for visualization with the further complication of the initial setup as a result. Additional methods for writing other file formats can be created to take advantage of high quality tools from external software packages, e.g. ParaView[10] which can visualize large datasets interactively.

[10]ParaView website: `http://http://www.paraview.org`

# 9 Experiments

The following is a description of the experiments, which are divided into demo applications, that are created to test and verify various aspects of the implementation. Unless stated otherwise, the implementations are compiled in Xcode 5 with the default compiler, Apple LLVM 5.0, and run on a MacBook Air '12 with OSX 10.9 Mavericks with an Intel Core i5 CPU at 1.8 GHz and 8 GB RAM.

## 9.1 Translation

A circular interface in a rectangular domain is translated. This demo will show that the interface is moved while the simplices of both the interior and the exterior are changed to allow for the movement of the interface. The velocity field is defined as

$$\mathbf{v}_t = (\alpha, \beta) \tag{7}$$

where $\alpha$ and $\beta$ are scalars. In this test, there are three movements, each of 120 steps. The first movement is with velocity $(8, 0)$. After that, the velocity is changed to $(0, 8)$ and at last, the velocity is changed to $(-8, -8)$. The expected result is that the circle (of constant radius) will move towards the right side of the domain, then down and then at an angle towards the starting point where it ends. The simplices of the phase (interior) are expected to change as the interface moves.

The second test is where a circular interface *and* the interior is translated. The interior triangles are therefore not expected to change at all during the simulation because their size and shapes are acceptable at time 0 and since the interior vertices are moved with the interface, this does not change. The exterior simplices must be removed and refined such that the interface can move. This demo will show that it is possible to use the movement step on a phase and not only the interface in applications where this is needed. The translation steps used are the same for the interface translation demo above.

## 9.2 Scaling

A circular interface in a rectangular domain is scaled outwards equally along both axes around origin. Each vertex on the interface therefore has a velocity vector (all with equal and constant size) starting from the center of the circle and ending at the vertex. This demo will show that the interface is refined when the distance from each vertex increases. The velocity field is defined as

$$\mathbf{v}_t = \alpha \frac{\mathbf{p}_t}{\|\mathbf{p}_t\|} \tag{8}$$

where $\alpha$ is a constant scalar. An application specific interpolator is used for interface vertices such that when the interface is refined, the shape remains round as opposed to looking more like a regular polygon with $n$ corners, where $n$ is the number of interface vertices at time step 0. This shows that custom application specific interpolators can be used in the implementation. The interior is expected to be refined as the radius of the circle increases and the exterior simplices are expected to be removed to make room for the growing circle.

## 9.3 Rotation

The first rotation demo is a rectangle in a rectangular domain, rotated around origin. Each vertex on the interface has a velocity perpendicular to the vector from the origin to the vertex. This test will show that the interface remains intact when the mesh changes and that the mesh operations work on vertices in general position. The second rotation test is a crescent shape in a rectangular mesh, rotated around origin. This test will show that the round part of the shape requires few changes while the open part of the crescent requires more changes. The velocity field is defined as

$$\mathbf{v}_t = \alpha \mathbf{p}_t^{\perp} \tag{9}$$

where $\alpha$ is a constant scalar.

## 9.4 Vortex

A rectangle in a rectangular domain is "twisted" and then untwisted in a vortex where the velocity of a vertex is the vector from origin to the vertex and scaled such that all velocities are with equal and constant size. The velocity field is defined as

$$\mathbf{v}_t = \alpha \frac{\mathbf{p}_t^{\perp}}{\|\mathbf{p}_t\|} \tag{10}$$

where $\alpha$ is a constant and the sign of $\alpha$ determines whether the interface is twisted left or right. The expected result is a twisted rectangle at time $t_{n/2}$ and an untwisted rectangle at $t_n$ similar to the one at $t_0$. This test uses centered difference method to calculate the new positions because it has a smaller truncation error.

## 9.5 Interpolation

A circular interface in a rectangular domain is scaled outwards equally along both axes around origin. The velocity field is defined by Equation 8. A 0-simplex attribute vector with real values is created and the values of the circular interface and interior 0-simplices are initialized to 0. As time progresses, the value of the vertices of the interface are interpolated linearly towards 1. When new vertices are inserted in the interior by loop subdivision, the values are interpolated linearly using the values of the subdivided triangle. The expected result is that the values of the vertices will increase linearly from 0 (at the initial radius) to 1 on the interface after the last time step. We cannot expect the result to be perfectly linear because of the non-linear shape of the interface. This test shows that an application specific interpolation method can be used for simplices that are not on the interface. Two tests with different triangle sizes are created to show that the non-linearity is approximated better with the smaller triangles.

## 9.6 Enright's Tests

In [6], two tests are used to determine how well the method introduced in the paper performs. The first test is the rotation of a Zalesak's disc, a circle with a

rectangle cut out in the bottom. The rotational center is below the circle. The expected result is that after one revolution, the shape of the disc is preserved. The second test is a circle that is placed with its center above the center of the domain. A stream function is given as

$$\Psi = \frac{1}{\pi} \sin^2(\pi x) \sin^2(\pi y) \tag{11}$$

and the velocity is then given as

$$\begin{aligned} \mathbf{v}_t &= \left( \frac{\partial \Psi}{\partial y}, -\frac{\partial \Psi}{\partial x} \right) & (12) \\ &= \left( \frac{1}{\pi} \sin^2(\pi x) \sin(2\pi y), -\frac{1}{\pi} \sin^2(\pi y) \sin(2\pi x) \right) & (13) \end{aligned}$$

where $(x, y) = \mathbf{p}$. The flow is reversed after a number of revolutions. A centered difference method is used and the test will be referred to as the swirling test. The expected result is that the shape of the sphere is preserved after the flow has been reversed.

## 9.7 Speedup

To measure the relative performance between the parallel algorithm and the sequential algorithm, we define the speedup factor $S(n)$ as

$$S(n) = \frac{t_s}{t_n}, \tag{14}$$

where $t_s$ is the computational time for the sequential algorithm and $t_n$ is the computational time for the parallel version using $n$ processes, with $n = \{1 \dots 4\}$ for this demo. When $S(n) = n$, linear speedup is obtained [13]. The processor utilization is defined as

$$U(n) = \frac{S(n)}{n}, \tag{15}$$

which gives 1 for linear speedup. For many algorithms, the sequential and parallel version is supposed to always return exactly the same result. This cannot be expected for DSC because local changes might propagate differently when the mesh is divided into submeshes. This also mean that the measured speedup depends on how the application behaves, e.g. how far the interface has to be moved, the quality that the mesh must have before it is acceptable, how many times new submeshes must be created etc. Because creating the submeshes is done sequentially, this will affect the speedup negatively. In a real application, the calculation of new positions (or velocities) might also be done sequentially, but in this example, the application is very simple and fast to compute. The first test is scaling of a circle where the interface will overlap with the submesh boundaries and linear speedup is therefore not expected because new submeshes has to be created, maybe multiple times. In the second test, four smaller circles are translated such that the interface is never on the submesh boundary. In this test, $n$ is fixed to 4. Here it is expected to get almost linear speedup because new submeshes are not needed as long as the DSC algorithm

gets enough iterations to completely move the interface vertices. It still has to create submeshes once per time step, so a speedup slightly less than linear is to be expected. Even though the results are not expected to be exactly the same, similar results (with regards to quality) is to be expected.

The demo is compiled and run in Visual Studio 2010 with version 2.9.1 of a prebuilt 32 bit pthread library[11] on a machine with Windows 7 and an Intel Core i5-3570K CPU at 3.40 GHz and with 16 GB RAM.

## 9.8 Quality

A circular interface in a rectangular domain is scaled outwards equally along both axes around origin. The velocity field is defined by Equation 8. The first test tries the four combinations of coarsening and refining the interface and interior. This will show that it is possible to have different behaviour in the exterior and interior. The second test uses a custom threshold quality measure where the refinement and coarsening of the interior and the interface depends on the $x$-coordinate of the center of the triangle (average of vertices). The leftmost part of the interface and the rightmost part of the interface are the end points of an interpolation of the threshold values for refinement and coarsening. The left side favours large triangle and the right side favours smaller triangles. Similar for interface edges. Because the threshold values are interpolates over the circle, triangles in the middle will have medium size. When the application changes the radius of the circle, the end points of the interpolation moves as well which means that the triangulation has to always change to adapt to the new interpolated values. Many small triangles and fewer larger triangles is expected.

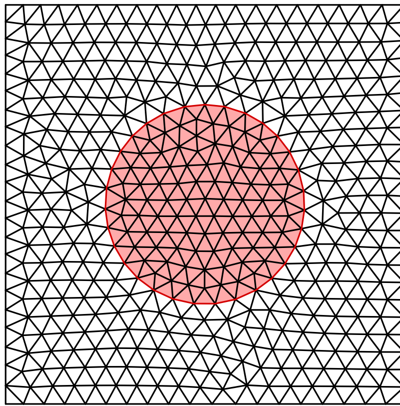
[11] `ftp://sourceware.org/pub/pthreads-win32/prebuilt-dll-2-9-1-release/`

# 10 Results

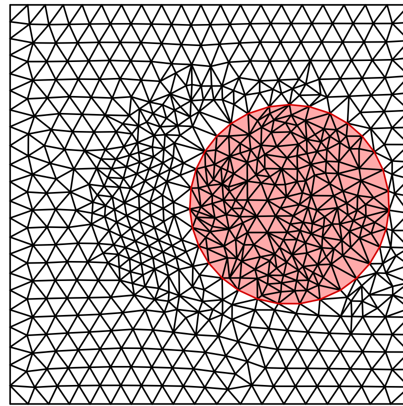
The following is the results of the experiments described in Section 9. A discussion of the results can be found in Section 11.

## 10.1 Translation

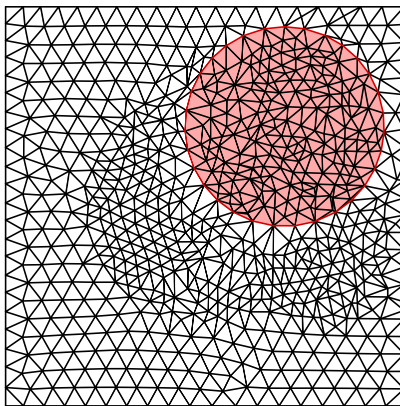
Selected frames from the translation of interface demo are shown in Figure 10.1 and frames from the translation of phase demo are shown in Figure 10.2. In the translation of interface demo, when the interface moves, a trail of new exterior triangles is left behind and the interior is changed as well. The new triangles seem to be slightly smaller than in the original mesh. The exterior triangles that are not close to the interface are unaffected. In the translation of phase demo, when the interface moves, a trail of new exterior triangles is left behind, but the interior is unchanged. The new exterior triangles are slightly smaller than in the original mesh. The exterior triangles that are not close to the interface are unaffected.



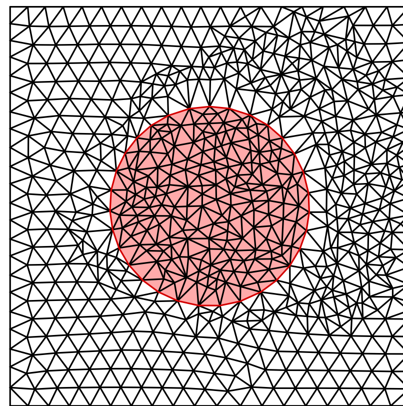
**(a)** Time step 0, 884 triangles



**(b)** Time step 120, 1251 triangles

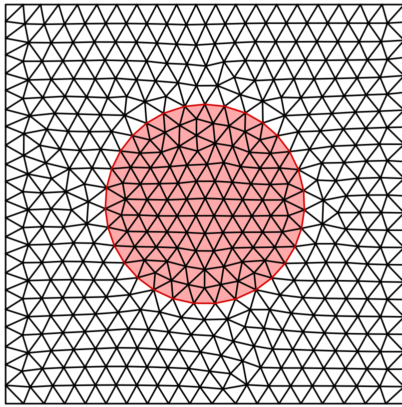


**(c)** Time step 240, 1367 triangles

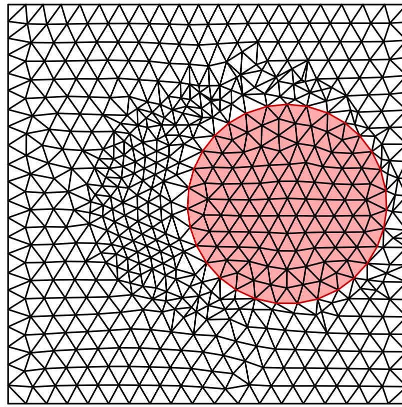


**(d)** Time step 360, 1305 triangles

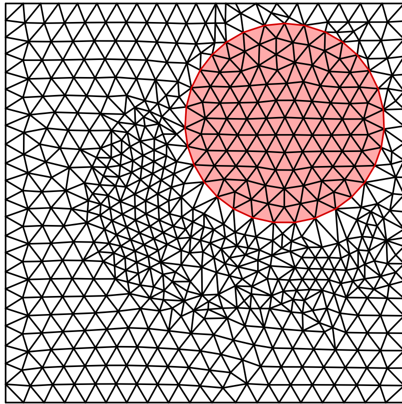
**Figure 10.1:** Selected frames from translation of interface demo.



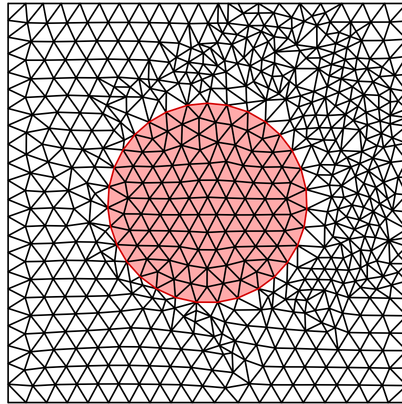
**(a)** Time step 0, 884 triangles



**(b)** Time step 120, 1119 triangles



**(c)** Time step 240, 1178 triangles

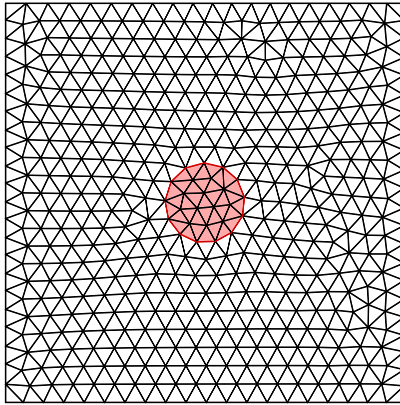


**(d)** Time step 360, 1199 triangles

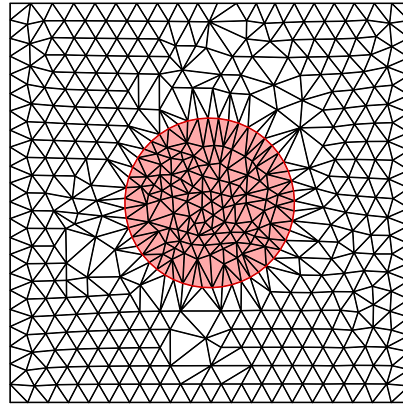
**Figure 10.2:** Selected frames from translation of phase demo.

## 10.2 Scaling

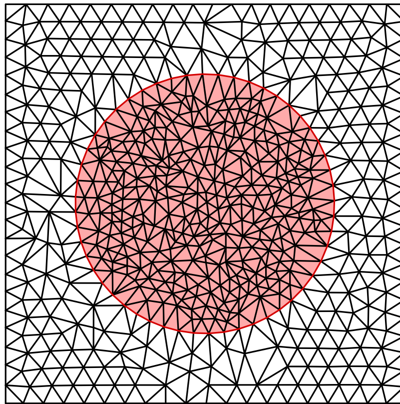
Selected frames from the translation of interface demo are shown in Figure 10.3. The interface of the circle is refined when the circle expands and it remains round. The inserted interior triangles are slightly smaller than the exterior triangles. The exterior triangles that are not close to the interface are unaffected.



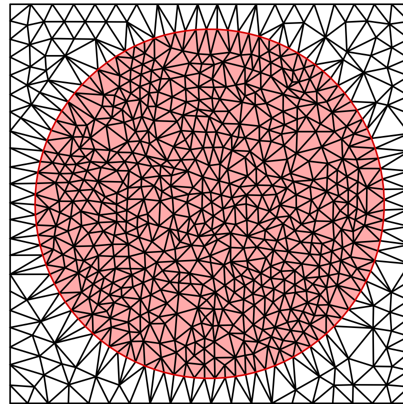
**(a)** Time step 0, 884 triangles



**(b)** Time step 90, 944 triangles



**(c)** Time step 181, 1144 triangles

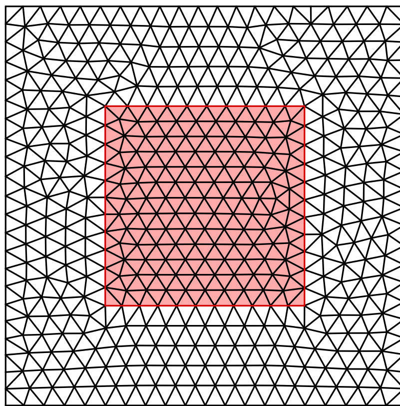


**(d)** Time step 270, 1442 triangles

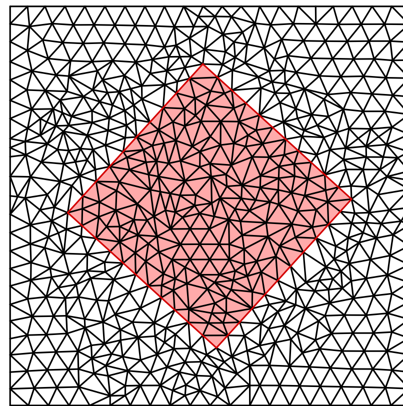
**Figure 10.3:** Selected frames from scale of circle demo.

## 10.3 Rotation

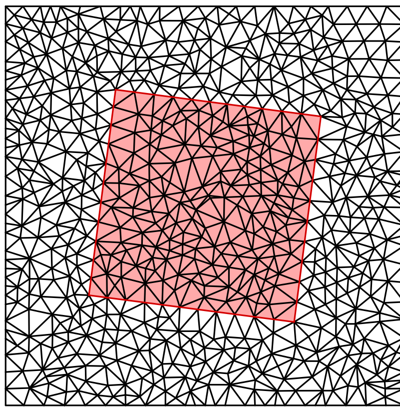
Selected frames from the rectangle rotation demo are shown in Figure 10.4 and frames from the crescent rotation demo are shown in Figure 10.5. In the rectangle demo the interior triangles are changed when the interface moves and many of the exterior triangles are moved as well. On the last frame, it seems like all exterior triangles are changed, including the corners that are furthest away from the interface. In the crescent demo, the exterior triangles are changed in the hole and when it moves, only the parts that are close to the hole are changed. The interior is changed such that the interface can move.



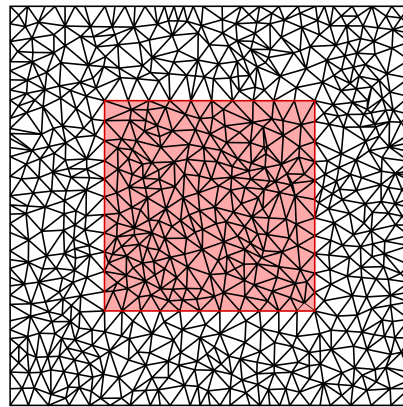
**(a)** Time step 0, 903 triangles



**(b)** Time step 50, 1179 triangles

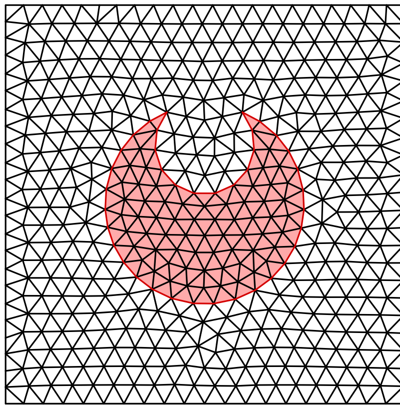


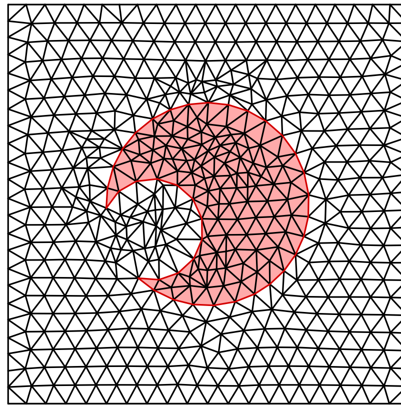
**(c)** Time step 275, 1345 triangles

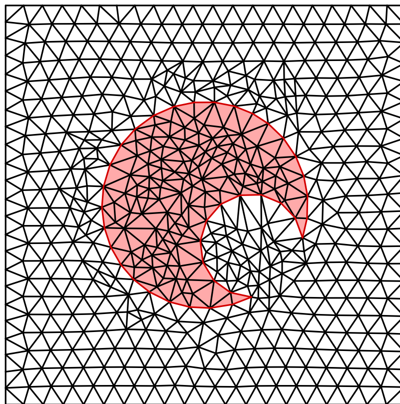


**(d)** Time step 377, 1363 triangles

**Figure 10.4:** Selected frames from rotation of rectangle demo.


**(a)** Time step 0, 890 triangles


**(b)** Time step 120, 1006 triangles


**(c)** Time step 240, 1050 triangles

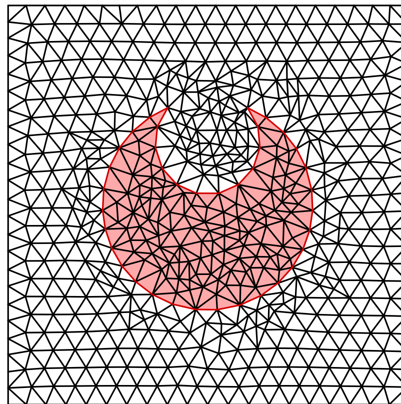

**(d)** Time step 377, 1094 triangles

**Figure 10.5:** Selected frames from rotation of crescent demo.

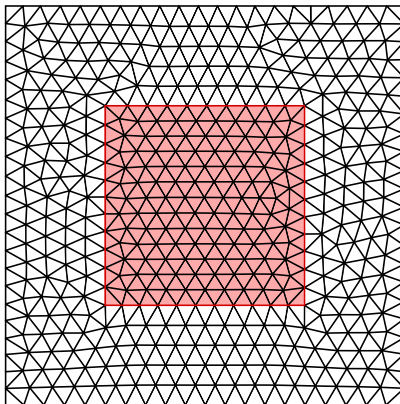
## 10.4 Vortex

Selected frames of the result of the vortex demo are shown in Figure 10.6. New edges are inserted to refine interface which allows the rectangle to twist. When the rectangle is untwisted, the edges are still more refined than in the input mesh at time 0. The areas computed are

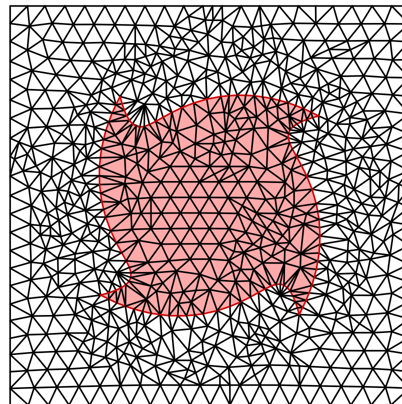
$$\begin{aligned}\text{area}(t_0) &= 10000.00\\ \text{area}(t_n) &= 9992.08\end{aligned}$$

where area($t$) is the area at time $t$ and $n$ is the last time step. This gives the ratio

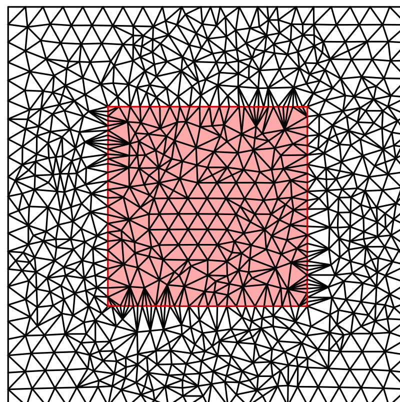
$$\frac{\text{area}(t_n)}{\text{area}(t_0)} = \frac{9992.08}{10000.00} \approx 1.00$$



**(a)** Time step 0, 903 triangles



**(b)** Time step 250, 1476 triangles

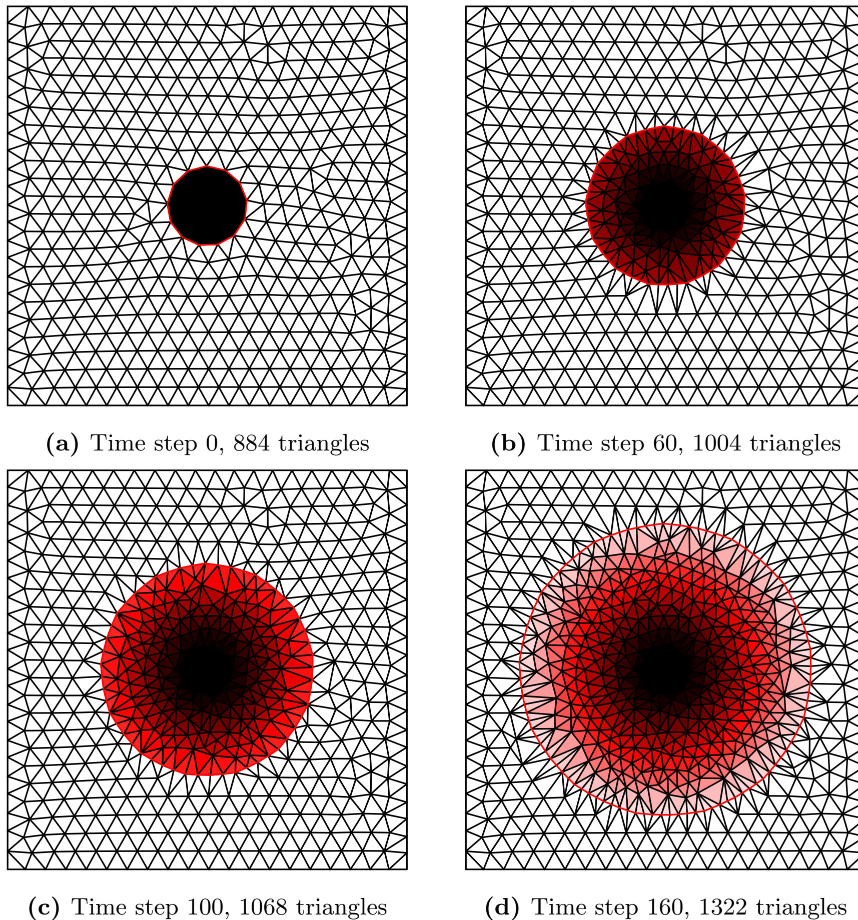


**(c)** Time step 500, 1579 triangles

**Figure 10.6:** Selected frames from vortex demo.

## 10.5 Interpolation

To visualize the results of the interpolation demo, the value of the vertices are used to colour the triangles by taking the average of the values of the corners of the triangles $l$ and converting them into an Hue Saturation Light (HSL) colour with 0 hue (red), 100% saturation and $l$ light. Frames from the demo are shown in Figure 10.7 for the larger triangles and in Figure 10.8 for the smaller triangles. A second rendering is shown in Figure 10.9 which uses Gouraud interpolation from matplotlib[12] of the vertex colours from blue to red through green and yellow.



**(a)** Time step 0, 884 triangles

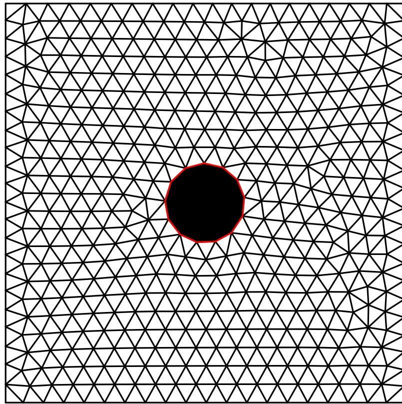
**(b)** Time step 60, 1004 triangles

**(c)** Time step 100, 1068 triangles

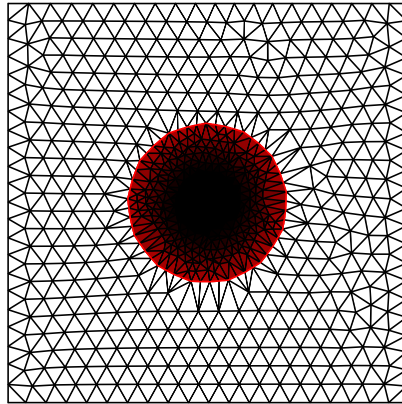
**(d)** Time step 160, 1322 triangles

**Figure 10.7:** Selected frames from interpolation demo with large triangles.

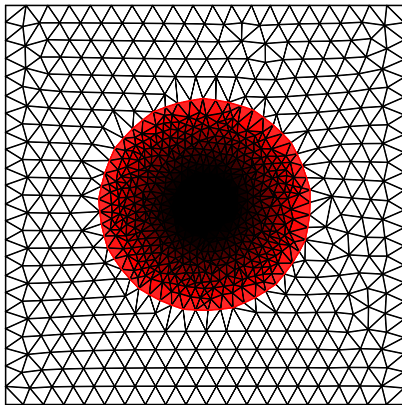
[12] matplotlib website: `http://matplotlib.org`



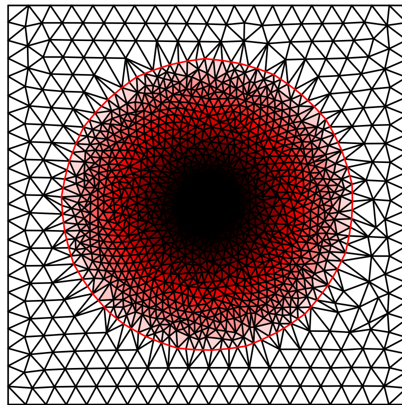
**(a)** Time step 0, 884 triangles



**(b)** Time step 60, 1298 triangles

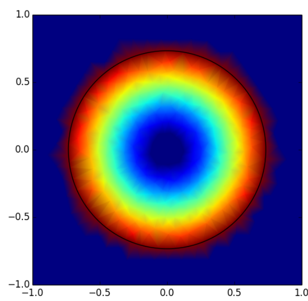


**(c)** Time step 100, 1680 triangles

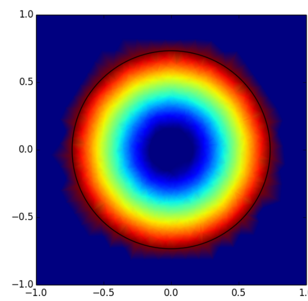


**(d)** Time step 160, 2602 triangles

**Figure 10.8:** Selected frames from interpolation demo with small triangles.




**(a)** Large triangles




**(b)** Small triangles

**Figure 10.9:** Time step 60 of interpolation demo with Gouraud shading. The black circle represents the interface. Figure 10.9a shows the result for the demo with large triangles and Figure 10.9b shows the result for the demo with small triangles.

## 10.6 Enright's Tests

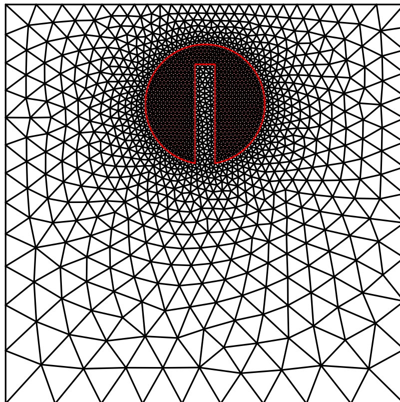
The result for the rotation of Zalesak's disc is shown in Figure 10.10. After one revolution, the triangles of the mesh seem to have a more consistent size. The disc looks identical to the disc at time 0. The result for the Enright test with the swirling circle is shown in Figure 10.11 with two tests, one with $2 \times 260$ frames and one with $2 \times 420$ frames. The first seem to preserve the circular shape, but the edge is not smooth anymore as it looks jagged. The second test does not look like the circle when in the last time step. it also has a jagged edge. The areas of the first swirling circle are

$$\frac{\text{area}(t_n)}{\text{area}(t_0)} = \frac{0.0701195}{0.0704815} \approx 0.9948639$$

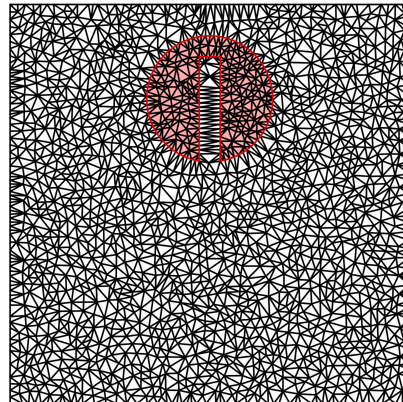
and the areas for the second swirling circle are

$$\frac{\text{area}(t_n)}{\text{area}(t_0)} = \frac{0.0683577}{0.0704815} \approx 0.9698673$$

Figure 10.12 shows the Enright swirl test with Zalesak's disc, but only going one way. The legs get very thin, but they are still separated and do not merge.

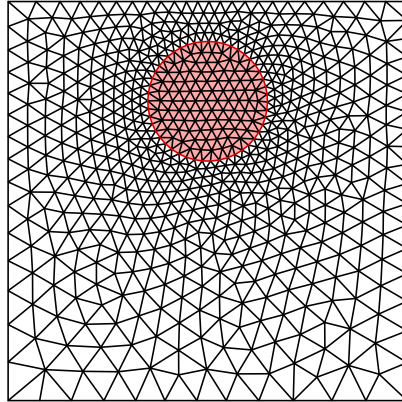


**(a)** Time step 0, 3264 triangles

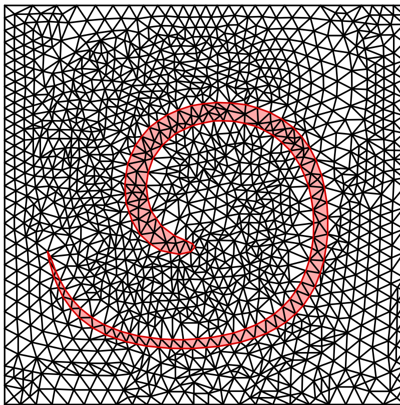


**(b)** Time step 377, 3754 triangles

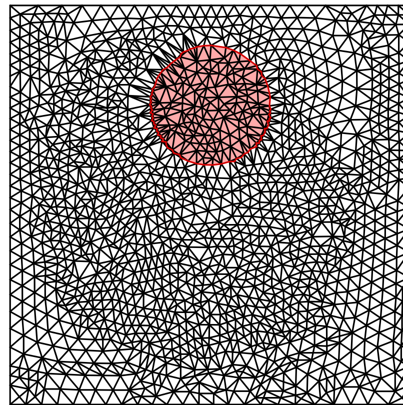
**Figure 10.10:** Selected frames from the rotation of Zalesak's disc demo. Figure 10.10a shows the input and Figure 10.10b shows the disc after one revolution.



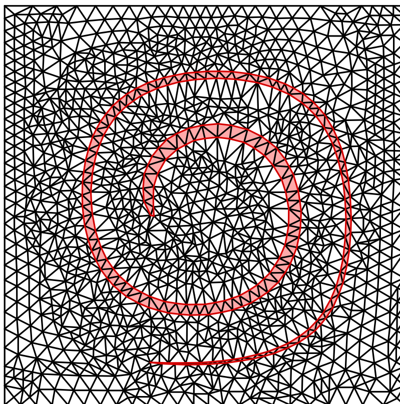
**(a)** Time step 0.



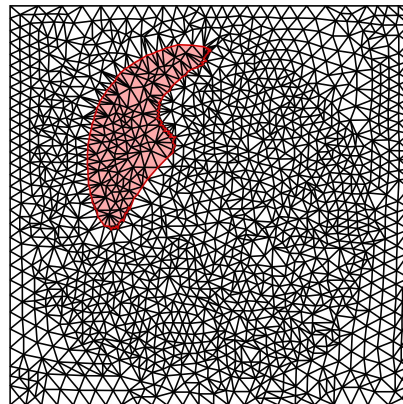
**(b)** First demo. Time step 260.



**(c)** First demo. Time step 520.



**(d)** Second demo. Time step 420.



**(e)** Second demo. Time step 840.

**Figure 10.11:** Frames from the Enright demo with the swirling circle. Figure 10.11a shows the input mesh. Figure 10.11b and Figure 10.11c shows the demo with $2 \times 260$ frames at frame 260 and frame 520. Figure 10.11d and Figure 10.11e shows the demo with $2 \times 420$ frames at frame 420 and frame 840.

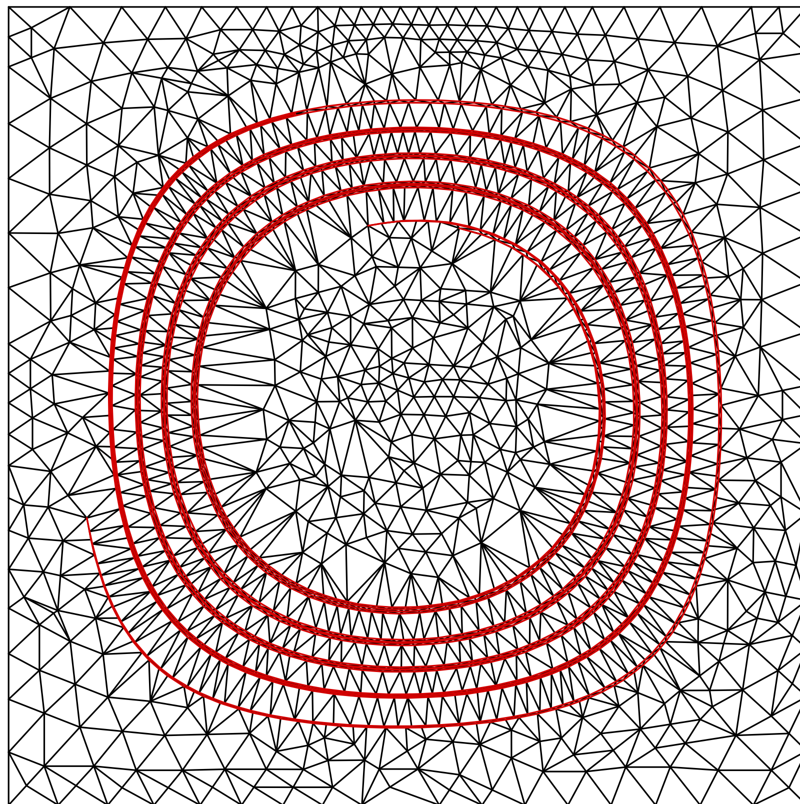


**Figure 10.12:** A frame from the Enright demo with a swirling Zalesak's disc at time step 800.

## 10.7 Speedup

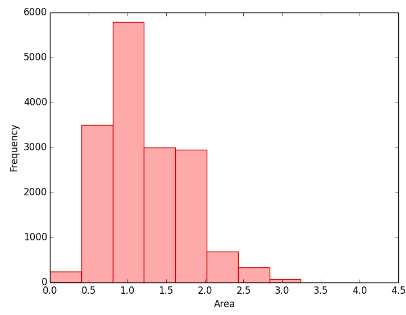
For the first test with the scaling of a circle, the computation times and speedups can be found in Table 10.1. Histograms of the area of the triangles and the quality using the quality measure in Equation 4 is shown in Figure 10.13 and Figure 10.14.

| $n$ | $t_n$ | $S(n)$ | $U(n)$ |
|---|---|---|---|
| $s$ | 179 s | 1 | 1 |
| 1 | 187 s | 0.96 | 0.96 |
| 2 | 122 s | 1.47 | 0.73 |
| 3 | 94 s | 1.90 | 0.63 |
| 4 | 77 s | 2.32 | 0.58 |

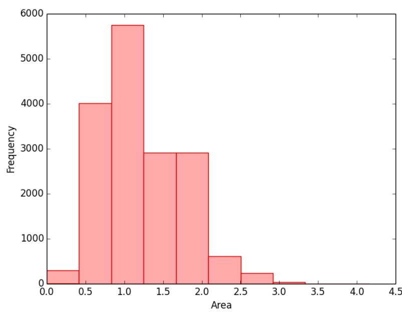
**Table 10.1:** Time and speedup for scaling.

For the second test with the translation of four circles, the running time and speedup are

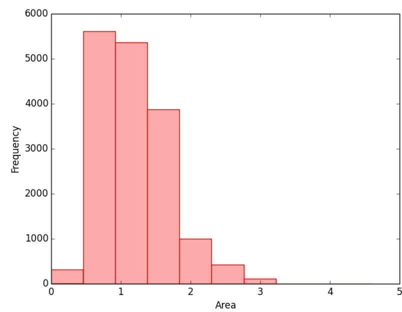
$$\begin{aligned} t_s &= 225 \text{ s} \\ t_4 &= 61 \text{ s} \\ S(4) &= \frac{225 \text{ s}}{61 \text{ s}} \approx 3.69 \\ U(4) &= \frac{3.688\ldots}{4} \approx 0.92 \end{aligned}$$



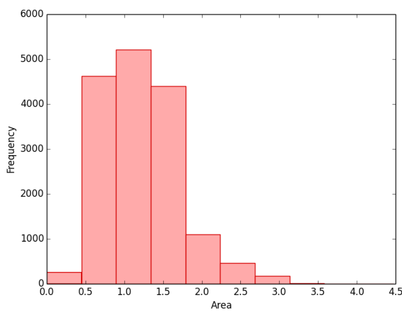

**(a)** Sequential, 28236 triangles



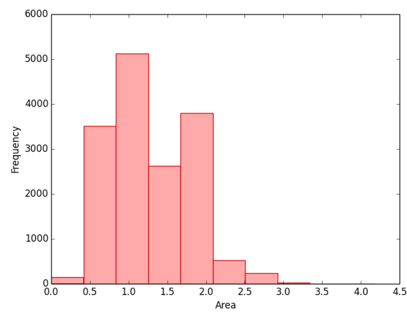

**(b)** 1 process, 28374 triangles




**(c)** 2 processes, 28389 triangles

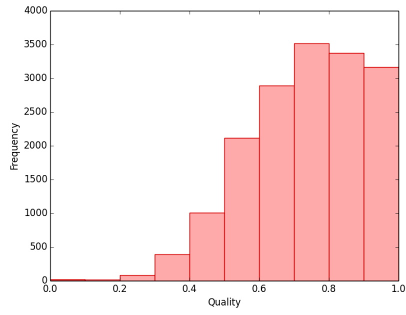



**(d)** 3 processes, 27936 triangles

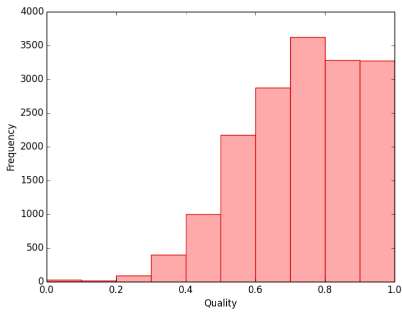



**(e)** 4 processes, 27670 triangles

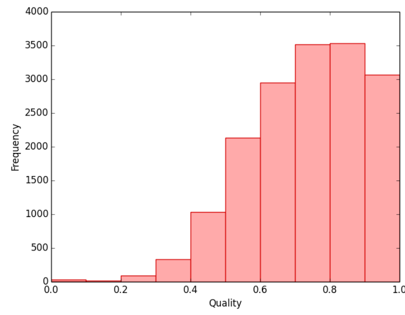
**Figure 10.13:** Area of the triangles in the scaling speedup demo.



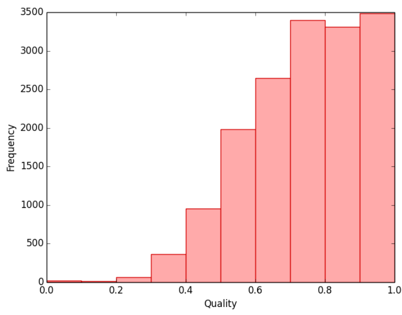

**(a)** Sequential, 28236 triangles



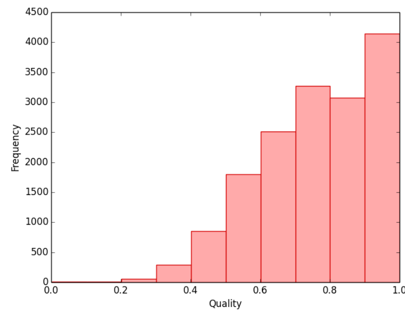

**(b)** 1 process, 28374 triangles




**(c)** 2 processes, 28389 triangles




**(d)** 3 processes, 27936 triangles



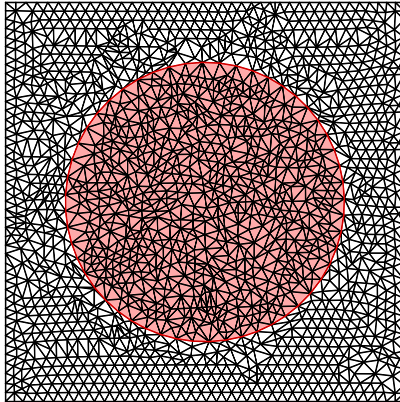

**(e)** 4 processes, 27670 triangles

**Figure 10.14:** Quality of the triangles in the scaling speedup demo.

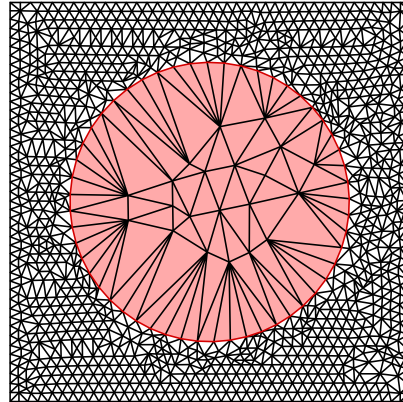
## 10.8 Quality

The frames at time step 30 for the quality demo with different combinations of triangle sizes for exterior and interior are shown in Figure 10.15 and selected frames from the quality demo with varying quality measures across a circle is shown in Figure 10.16. Figure 10.17 shows the areas of the triangles from the quality demo with varying quality measures in a histogram.

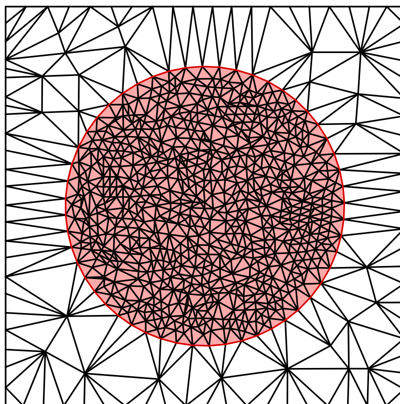
In the demo with different combinations of triangle sizes, small triangles are seen in phases that are supposed to be refined and larger triangles are seen in phases that are supposed to be coarsened. In the coarsened phases, some triangles are very sharp. In the demo with varying quality measures there are larger triangles on the left side and smaller triangles on the right side. Near the interface on the right side, some triangles are relatively small. The interface is more refined on the right side than on the left. In the histogram of the areas, there are many triangles of small size and fewer of larger size. The number seems to decay exponentially.



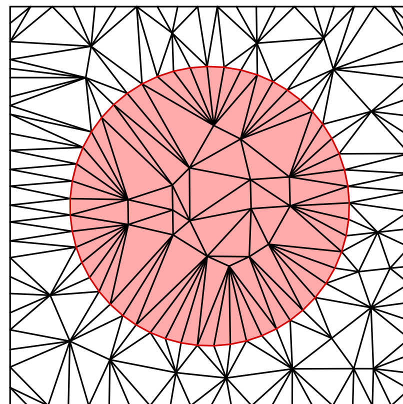
**(a)** Refined exterior, refined interior, 3409 triangles



**(b)** Refined exterior, coarse interior, 2146 triangles

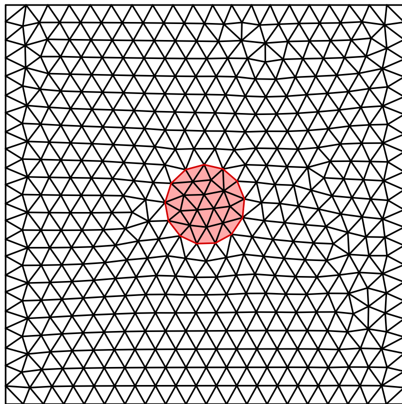


**(c)** Coarse exterior, refined interior, 1566 triangles

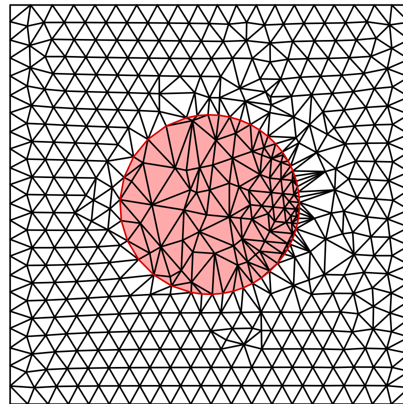


**(d)** Coarse exterior, coarse interior, 262 triangles

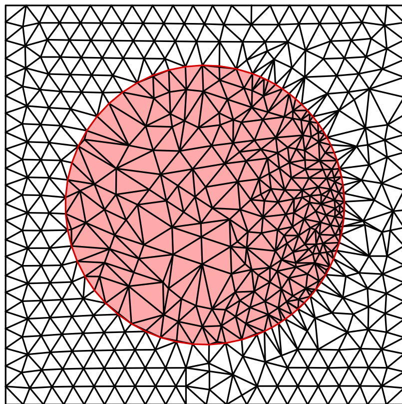
**Figure 10.15:** Frames at time step 30 of quality demos.



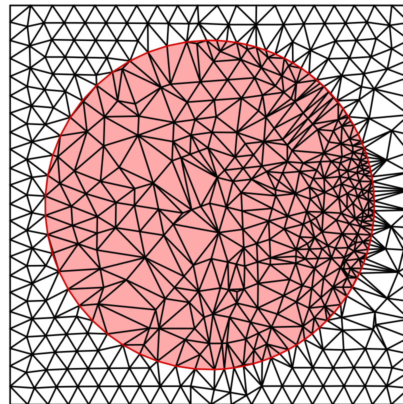
**(a)** Time step 0, 884 triangles



**(b)** Time step 40, 964 triangles



**(c)** Time step 80, 1026 triangles



**(d)** Time step 100, 1074 triangles

**Figure 10.16:** Selected frames from quality demo with varying quality measures.

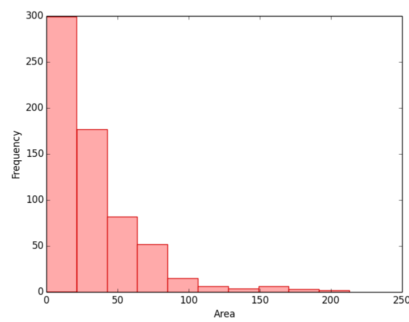



**Figure 10.17:** Area of triangles in frame 100 of quality demo with varying quality measures.

# 11 Discussion

The following is a discussion of the results in Section 10 with comparisons to the expected result explained in Section 9.

**Translation.** The motion of the circles is as expected in both translation tests and the first and last positions are the same. When the interface moves, the triangles behind the circle will get thinner and since the mesh operation used for optimization (edge flip) cannot be used here, the threshold for loop subdivision has been lowered slightly to create new triangles faster (otherwise the triangles would be even thinner). This means that the new triangles will be smaller. If there had been a smoothing method implemented, the thing triangles could possibly be avoided because it would move the vertices closer to the interface and make triangles less thin. The test with the moving interior correctly moves the interior without changing it because the sizes of the triangles remain constant and are not too small and not too large for the coarsening or the refinement operations to be called.

**Scaling.** The refinement of the interface of the circle uses a custom interpolator and this correctly moves the new vertices because the circle remains round. Here it is possible to see what happens when the threshold for loop subdivision is not lowered as they were in the translation demos. On some frames the triangles become very thin until the area is large enough for loop subdivision to be called. Also here smoothing could help to move some vertices closer to the interface to make the triangles less thin. In some applications, small triangles close to the interface is preferred, and in these cases, the problem might disappear because loop suvdivision is supposed to make small triangles close to the interface and the long, thin triangles are avoided.

**Rotation.** Both rotation demos behaves as expected. The interface is not refined at any time because the the distance between the vertices remains constant (and the lengths at time 0 are within the tolerated bounds).

**Vortex.** The rectangle in the vortex demo comes back to its original shape and the area is almost the same, although a little is lost which can be attributed to linear insertion of vertices on the interface and errors in the centered difference method. The number of triangles has increased because the interface has to be refined to allow for the shape, but it is not coarsened when it comes back because a coarsening operation would also coarsen it at its extreme position at $t_{n/2}$. One could create a coarsening operation that can only be used when the interface is (almost) a straight line, or maybe large deformations can be detected and used to decide when and where coarsening should be used.

**Interpolation.** The interpolated values behave as expected. Because a linear interpolator is used, only an approximation can be expected. The approximation when using larger triangles is not as good as the one with smaller triangles. The red colour in Figure 10.9 which overflows the circle is because the exterior triangles which has at least one vertex on the boundary will get the red colour

from the interface. This is a problem with the shading method used and the library for rendering them, and not the interpolation.

**Enright's Tests.** The rotation of Zalesak's disc preserves the shape of the disc as expected. The first swirl test preserves the circles overall shape, but the jagged edges can presumably be attributed to the linear insertion (and removal) of vertices. The second test does not preserve the circles overall shape, and it would seem that the centered difference method is not good enough for the chosen step size and number of steps. The swirling of the more complicated shape, namely Zalesak's disc, shows that the implementation does not fail with more complex movement, but another interpolation method and numerical method must be chosen to get more accurate results. The implementation supports changing these from the application level.

**Speedup.** The resulting meshes of the scaling demo are reasonable similar, but the demo is also fairly simple. The speedups do not utilize the CPU very well. Submeshes has to be created at least two times because some of the interface vertices are locked the first time. The second time only a few vertices has to be moved, but the algorithm will still compute qualities for all simplices, but since only a few of them will change, time is mostly wasted. This observation is strengthened by the fact that a much better speedup is gained when no vertices are on the submesh boundary as in the second test with the translation of circles. In this test, the submeshes only has to be created once, which means that less time is wasted on computing qualities. If the quality in the first demo is allowed to be low, the speedup is even smaller because the amount of work there has to be done is smaller and the time spend on measuring quality is too large compared to the actual work done using mesh operations. On the other hand, when the mesh must have greater quality, the speedup is better because the processes have more work to do. The problem with the scaling demo is that there is in general not much work to be done. The changes are near the interface which means that most of the entire mesh is unchanged. It is therefore reasonable to believe that an application with more changes (and maybe with an implementation of smoothing) would give a better speedup, but due to the time constraint, this has yet to be attempted. Having a test machine with more than four cores could also be interesting for testing the behaviour.

**Quality.** The triangle sizes behave as expected in both tests. They could both probably benefit from a smoothing step because some vertices are very close which makes some of the triangles very sharp. A test where the quality depends on the distance to the interface would be useful because that is desired in some real applications. Due to the time constraint, this has yet to be done. A problem with it is that the distance to the interface is not known when the submeshes are created because the closest might be in a neighbour submesh. A solution could be to compute them before the submeshes are created, but this would only give an approximation because the interface is moved. A simpler solution could be to assume that the closest interface is in the submesh, but that might not work well for all applications. In Figure 11.1 frames from the original 2D DSC implementation are shown. Both the interior and exterior are very coarse. This behaviour is also possible in the implementation proposed in

this thesis, as shown in Figure 10.15d, but the quality can also be changed to be more refined which is better in some applications.

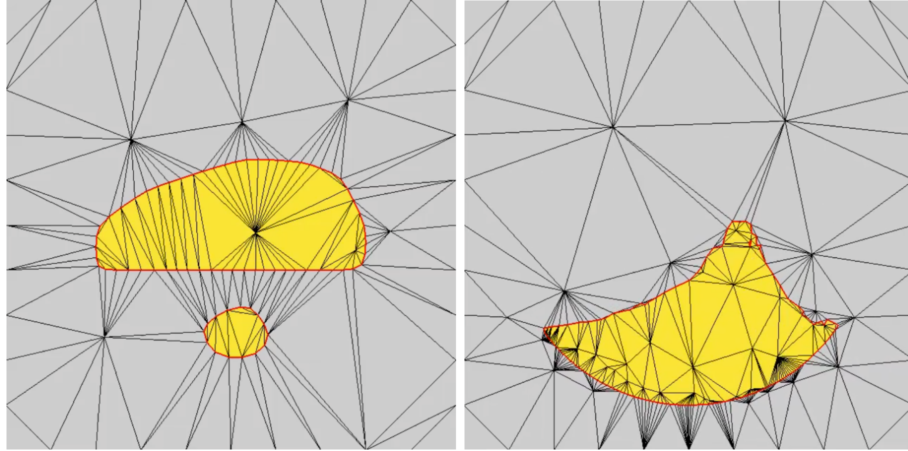


**Figure 11.1:** Two frames from the original 2D DSC implementation.

Most of the demo applications give the expected results, but a smoothing step would probably make some of the meshes more regular. Especially now that the operations are parallelized, the cost of smoothing might be worth the improvements in quality that it gives. Implementations of more advanced applications, e.g. a fluid simulation or a sculpting tool, would show if the framework is flexible enough for real use, but the different demo applications do have a variety that demonstrates the flexibility in use of quality measures, interpolators etc. More functionality needs to be implemented and tested, for example merging and splitting of phases (topology control) which is necessary for fluid simulations. Most of the mesh operation implementations done in this thesis will keep the interface intact, but they could be changed to ignore the interface restrictions by introducing a simple boolean parameter such that both types of behaviour is possible.

# 12 Conclusion

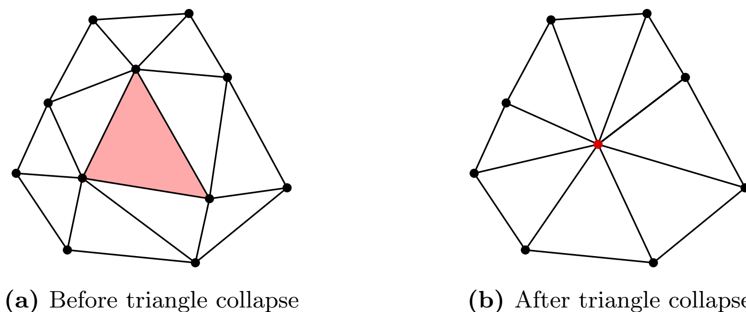
A design for a parallel implementation for Deformable Simplicial Complex (DSC) is proposed. It has been shown that the implementation is highly configurable because quality measures, iteration stop criteria, interpolators etc. can be changed from the application level. Mesh operations are expressed using topology algebra which means that it is possible to change the mesh data structure by implementing simplex relations and operations on top of it. This can be done without reimplementing the DSC algorithm. Dealing with the submesh boundary in the mesh operations is simple and does for the most part only require about two additional checks before the operation can be executed. A default implementation of the DSC algorithm is provided, but if an application has other needs, a new one can be implemented while again preserving the parallel aspect. The level of modularity means that even if some parts need to be replaced, other parts can be reused. Several demo applications have shown the flexibility of the implemented framework. A scaling demo gained a speedup of 0.96 for 1 process, 1.47 for 2 processes, 1.90 for 3 processes and 2.32 for 4 processes. This gives CPU utilizations of 0.96, 0.73, 0.63 and 0.58. It has been shown using another demo, which gains a speedup of 3.69 with 4 processes (CPU utilization of 0.92), that the bottleneck is when there are too few changes in the mesh per time step. In that situation, the computation of mesh quality using quality measures dominates the result because submeshes has to be created at least two times when the interface intersects the submesh boundary. It has been shown that the quality of the sequential algorithm and the parallel algorithm are comparable.

A framework that is easy to configure and use for simulations, and other similar types of applications, is important for many reasons. It makes the time from an idea to a prototype shorter, but it also makes it possible to fine-tune behaviour for advanced applications. For educational purposes it is also very important that the framework is easy to install because otherwise it will give students less time for actual learning and will cause frustration. This implementation still has two dependencies that ought to be removed, but compared to the previous implementations, this is a definite improvement. The increase in speed that the parallelism gives, means that the results of a simulation can be computed quicker and there is more time for experimenting or creating new simulations, but it can also be used to do computations with a finer granularity without exceeding a time budget. This can lead to more accurate results which is in some fields very important.

# 13 Further Work

The following is a description of some of the things that can be worked on to improve the implementation.

**Merging of phases.** The implementation does not support merging of two phases because all the implemented simplex operations keep the interface intact (with the exception of edge split and edge collapse, but these are so far only used to refine and coarsen an interface). Triangle collapse, see Figure 13.1, could be useful because it can remove small triangles between two phases. An example of when merging of two phases if necessary is in a liquid simulation when a droplet is falling down and absorbed in the liquid. Triangle collapse may be possible to implement using edge collapse twice, once on one of the edges of the triangle, and then on the remaining edge after the first collapse (this could also mean that triangle collapse is entirely unnecessary and that edge collapse is enough).



**(a)** Before triangle collapse **(b)** After triangle collapse

**Figure 13.1:** Example of triangle collapse. The input 2-simplex is shown in red in Figure 13.1a and the new vertex inserted after triangle collapse is shown in red in Figure 13.1b. The input simplex and the incident triangles are deleted and the triangles that shared a vertex with any of the vertices of the input triangle are updated to use the new vertex.

**Splitting of phases.** A method for splitting an interface is not implemented. This is necessary for example in liquid simulations when a droplet is about the separate itself from another part of the fluid.

**Smoothing.** DSC typically has an additional step called smoothing. This step moves vertices to make them more evenly distributed. An example of a smoothing method is Laplacian smoothing.

**Alternative to simplex sets.** To represent simplex sets, the set implementation from the C++ standard library, `std::set`, is used. While they ensure that the simplex set is an actual set (no duplicate members), insertion is slower than regular array like data types. Especially when inserting a large amount of simplices that are known to be different, the sets will for each insertion check whether the element is already a member. If an array data structure was used instead, some mesh operations will fail because they rely on the fact that there are no duplicates. It may be worth investigating a possible compromise. A

linear data structure would also make it more easy to move the data to other memory units (e.g. the memory of a GPU).

**Alternative attribute vectors.** The attribute vectors for 1-simplex and 2-simplex uses the map implementation, `std::map`, from the C++ standard library. A linear data structure would also make it more easy to move the data to other memory units (e.g. the memory of a graphics card).

**Mesh operations without coordinates.** Some of the mesh operations use coordinates even though they are valid in a purely topological setting. Loop subdivision is an example where the implementation uses the coordinates to calculate new coordinates for the inserted vertices. This should be handled by an interpolator[13].

**Further reduction of dependencies.** The implementation depends on PolyMesh from OpenTissue, and OpenTissue depends on the Boost vector library. It would be useful to have a mesh data structure implementation and a vector implementation included in the library to remove all dependencies. This would make the framework easier to setup which is sometimes a huge time sink and can be frustrating for many new users.

**Three dimensions.** While two dimensional simulations are fine, supporting three dimensions is often also desirable. The mesh operations described in this thesis cannot all be generalized to three dimensions, so new operations have to be developed and the submesh boundary considerations (the way to think about them) are applicable to 3D. The submeshing methods should also be changed to work for three dimensions, or maybe a new one that will work in both cases. Otherwise, the parallel DSC algorithm ought to work with three dimensions.

**Substitution for ear clipping.** The coarsening method used is local retriangulation with ear clipping. Ear clipping can create very thin slivers and it may therefore be better to use a Delaunay triangulation method instead. In general, local retriangulation is difficult to control because it removed multiple triangles (and not just the one that was chosen), and therefore triangle collapse could be an alternative altogether.

**Culling of simplices.** When submeshes are created twice, often only a few simplices need to be changed the second time. A method for finding these simplices, e.g. by only considering those that were close to the submesh boundary, a better speedup might be gained because as the experiments have shown, the bottleneck is when there is a need to create submeshes multiple times.

**Reimplementation of applications.** An implementation of the simulations that the old DSC implementations were used for, e.g. fluid simulation, would provide a basis for comparison between them, both regarding quality and accuracy, but also speed.

[13]An interpolator is possible in the current implementation, but it will have to overwrite the coordinates that the mesh operation has inserted.

## References


[1] David A. Patterson and John L. Hennesy. *Computer Organization and Design*. Morgan Kaufmann, 2011.

[2] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. A finite element method for animating large viscoplastic flow. *ACM Trans. Graph.*, 26(3), 2007.

[3] Andrey N. Chernikov and Nikos P. Chrisochoides. Practical and efficient point insertion scheduling method for parallel guaranteed quality delaunay refinement. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 48–57, New York, NY, USA, 2004. ACM.

[4] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN ENGINEERING*, 46:1101–1125, 1999.

[5] Mark de Berg, Otfried Chong, Mark van Kreveld, and Mark Overmars. *Computational Geometry*. Springer, 2008.

[6] D.P. Enright, Stanford University. Program in Scientific Computing, and Computational Mathematics. *Use of the Particle Level Set Method for Enhanced Resolution of Free Surface Flows*. Stanford University, 2002.

[7] G. Irving, J. Teran, and R. Fedkiw. Invertible finite elements for robust simulation of large deformation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, pages 131–140, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association.

[8] Mark Viinblad Jensen. Increasing quality and improving interfaces of the deformable simplicial complex library, 2012. Project report from Department of Computer Science, Copenhagen University.

[9] Marek Krzysztof Misztal. *Deformable Simplicial Complexes*. PhD thesis, Technical University of Denmark (DTU), Denmark, 2010.

[10] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.

[11] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[12] Jonathan Richard Shewchuk. What is a good linear finite element? - interpolation, conditioning, anisotropy, and quality measures. Technical report, In Proc. of the 11th International Meshing Roundtable, 2002.

[13] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice-Hall, 1999.