High-Quality Simplification and Repair of Polygonal Models

Dissertation

zur Erlangung des Doktorgrades (Dr. rer. nat.)
der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Dipl.-Math. Pavel Borodin

aus Moskau

Bonn, Mai 2009

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

1. Gutachter: Prof. Dr. Reinhard Klein
2. Gutachter: Prof. Dr. Andreas Weber

Tag der mündlichen Prüfung: 24. August 2009

Diese Dissertation ist auf dem Hochschulschriftenserver der Universitäts- und Landesbibliothek Bonn (http://hss.ulb.uni-bonn.de/diss_online) elektronisch publiziert.

Erscheinungsjahr: 2009

Acknowledgements

The work presented in this thesis has been produced within the scope of the Computer Graphics group at the Institute of Computer Science II of the University of Bonn.

I wish to thank all the people who were directly or indirectly involved in the creation of this work. First of all, I want to express my deepest gratitude to Professor Reinhard Klein, the leader of the Computer Graphics group and my scientific adviser, without whose patience and support this thesis would not have been possible.

Furthermore, I thank all members of the Computer Graphics group, especially those with whom I had a pleasure of joint research and writing papers: Marcin Novotni, Michael Guthe, Gabriel Zachmann and Alexander Greß. For the same reason my thanks belong to Stefan Gumhold.

Our secretaries Simone von Neffe and Regina Haverkamp play an important role in the life of our group, always ready to help and to sort out any administrative issues, for what I am very grateful to both of them.

Finally, I want to thank DaimlerChrysler AG, the Stanford 3D Scanning Repository, the Digital Michelangelo Project, the Virtual Try-On project and Michael Beals, who created and/or provided many of the polygonal models used in my research and, consequently, in this thesis.

Abstract

Because of the rapid evolution of 3D acquisition and modelling methods, highly complex and detailed polygonal models with constantly increasing polygon count are used as three-dimensional geometric representations of objects in computer graphics and engineering applications. The fact that this particular representation is arguably the most widespread one is due to its simplicity, flexibility and rendering support by 3D graphics hardware. Polygonal models are used for rendering of objects in a broad range of disciplines like medical imaging, scientific visualization, computer aided design, film industry, etc.

The handling of huge scenes composed of these high-resolution models rapidly approaches the computational capabilities of any graphics accelerator. In order to be able to cope with the complexity and to build level-of-detail representations, concentrated efforts were dedicated in the recent years to the development of new *mesh simplification* methods that produce high-quality approximations of complex models by reducing the number of polygons used in the surface while keeping the overall shape, volume and boundaries preserved as much as possible.

Many well-established methods and applications require "well-behaved" models as input. Degenerate or incorectly oriented faces, T-joints, cracks and holes are just a few of the possible degenaracies that are often disallowed by various algorithms. Unfortunately, it is all too common to find polygonal models that contain, due to incorrect modelling or acquisition, such artefacts. Applications that may require "clean" models include finite element analysis, surface smoothing, model simplification, stereo lithography. *Mesh repair* is the task of removing artefacts from a polygonal model in order to produce an output model that is suitable for further processing by methods and applications that have certain quality requirements on their input.

This thesis introduces a set of new algorithms that address several particular aspects of mesh repair and mesh simplification. One of the two mesh repair methods is dealing with the inconsistency of normal orientation, while another one, removes the inconsistency of vertex connectivity. Of the three mesh simplification approaches presented here, the first one attempts to simplify polygonal models with the highest possible quality, the second, applies the developed technique to out-of-core simplification, and the third, prevents self-intersections of the model surface that can occur during mesh simplification.

Contents

Part I

Introduction and basics

Chapter 1

Introduction

This chapter familiarizes the reader with the areas of research presented in this thesis. The problems treated in this work are discussed in section 1.1, while section 1.2 provides a brief overview of our contributions to the solution of these problems.

1.1 Motivation and problem statement

One of the main goals of computer graphics applications and research is to provide realistic description and visualization of 3D objects. According to recent trends and development in this field, highly complex objects described by huge amounts of data have become common.

The rendering performance of common desktop PCs have grown drastically in the last few years, and the bandwidth of networks increases rapidly as well. However, these both factors are still the major constraints determining the desirable level of complexity of 3D objects to be worked with. In order to provide a flexible solution to cope with this problem, a number of methods have been suggested.

Mesh simplification techniques facilitate the reduction of complexity while optimally approximating the original model in terms of an error.

Level-of-detail representations consist of several meshes with varying degree of simplification, and thereby enable the use of desired level of complexity: the objects lying farther or closer to the observation point in the scene are represented with more or less detail content, respectively.

Progressive representations consist of models with detail content encoded in a progressive manner in the sense of an approximation error, which enables transmission of complex models by first transmitting a coarse representation

and subsequent successive addition of detail. Hence, progressive representations may be interpreted as special level-of-detail representations as well.

A common issue in mesh simplification is whether or not to retain the topology of original models. In some cases it is desirable to join the disconnected parts of an object at a certain simplification level, thereby resulting in a better approximation of the original model. In order to facilitate such topology modification, the *vertex contraction* operator is commonly used. This simplification technique allows for unifying vertices not necessarily connected by an edge, thus enabling topology changes during the decimation process. However, in some applications we do not want to unify vertices lying in arbitrary regions of the mesh in order to retain the topological structure of the model – in this case it is common to use the *edge collapse* operation, which only unifies vertices lying on a common edge.

In many cases these well-established decimation techniques are entirely sufficient. At the same time, for some classes of models and in some certain applications, the methods proposed in this thesis can deliver much better results.

On the other hand, a common problem of geometric modelling tools is the generation of consistent three-dimensional meshes. Models from different sources like remote sensing, medical scanning, CAD and even scientific computing contain degenerate or incorectly oriented faces, T-vertices, narrow gaps and cracks. Applying well-established methods and applications, including finite element analysis, surface smoothing, model simplification, stereo lithography and milling, to meshes with such degeneracies often results in severe artefacts.

The industrial relevance of this problem is emphasized by the fact that as an output of most of the commercial CAD/CAM and other modelling tools, the user usually gets consistent meshes only for separate polygonal patches as opposed to the whole mesh. *Mesh repair* aims at creating a similar model but without its flaws. To achieve this goal the topology and sometimes also the geometry of the given mesh has to be modified.

1.2 Contributions and thesis structure

This thesis describes two approaches to repair polygonal meshes, which contain two particular types of artefacts due to incorrect modelling or acquisition, as well as three methods related to three different aspects of mesh simplification. Below, we will provide a brief background of the problem that each of these algorithms attempts to solve and describe their individual contributions.

All methods presented in this thesis were developed and pre-published at different international computer graphics conferences between the years 2001 and 2004.

1.2.1 Mesh repair

1.2.1.1 Consistent orientation of normals

In many areas of computer graphics, such as real-time rendering, mesh processing, etc., models are assumed to consist of correctly oriented polygons. Incorrect orientation of model's primitives could cause such methods to produce severe artefacts. However, many geometric modelling tools (such as CAD systems) pay little attention to the orientation of the normals, when exporting polygonal models.

We propose a method that can consistently orient all normals of any mesh (if it's possible at all), while ensuring that most polygons are seen with their front faces from most viewpoints. Our algorithm combines the traditional *proximity-based* approach with our new *visibility-based* approach.

We first divide the model into a set of manifold surface patches and consistently orient the polygons within each patch. Then, we determine the *proximity* of the patches to each other across their common boundaries and approximate the *visibility* for each patch regarding all possible viewpoints. Based on these values we orient the patches so that consistency between the ones with close boundaries is maximized, and the visible surface of as many patches as possible is seen with their front faces from as many viewpoints as possible.

We have tested our method with a large suite of models, many of which are from the automotive industry. The results show that almost all models can be oriented consistently and sensibly using our algorithm.

1.2.1.2 Progressive gap closing

Many meshes contain degenerate faces, T-vertices, narrow gaps and cracks. Due to lack of consistent connectivity information, many rendering and processing methods produce undesirable artefacts, if applied to such meshes. Several approaches exist aimed at generation of topologically connected meshes. We propose to interpret this issue as a mesh boundary simplification task.

By using a *vertex contraction* operation we are able to join unconnected regions of the mesh. In addition to it and the usual *edge collapse* operation, we introduce a new *vertex-edge contraction* operation. By utilizing this operator to simplify the boundaries, we manage to remove the artefacts in 3D models, such as T-vertices, degenerate triangles, etc. Furthermore, by applying the operator according to an increasing error, we successively close the gaps and holes in the model.

This provides extra support for closing gaps and stitching together the boundaries of triangle patches lying in near proximity to each other. In our method, the decimation process is error controlled and conducted in a progressive manner in terms of the error. Therefore, the user is enabled to visually inspect and interactively influence the procedure.

1.2.2 Mesh simplification

1.2.2.1 High-quality simplification

In works by Garland/Heckbert and Popović/Hoppe, the traditional *edge collapse* operation (also known as *edge contraction*) was generalized to *vertex contraction*, which allows for topology modifications during the decimation. Because of its simplicity, the vertex contraction operation became very popular in mesh simplification.

The vertex contraction facilitates the joining of originally disconnected regions of the mesh by contracting vertices lying in different connected components of the model. While this operation provides considerable topological flexibility during the mesh simplification, in some cases it is not general enough to connect close or even intersecting surfaces with small error early in the simplification

We propose the use of the *generalized pair contractions*: contraction of a vertex with another vertex, an edge or a triangle and also contraction of two edges. These operations have several advantages over standard vertex contraction. They allow to repair cracks and self-intersections and to sew unconnected components with lesser error.

In addition to its ability to repair meshes in an intuitive and efficient way, simplification using generalized pair contractions often produces much better decimation results than previous simplification techniques. Furthermore, our algorithm is particularly useful for the simplification of the models consisting of a large number of unconnected parts, such as industrial machines, generated by CAD/CAM and other geometric modelling tools.

1.2.2.2 Out-of-core simplification

A general strategy for out-of-core simplification is to split the model into smaller blocks, simplify these blocks and stitch them together for further simplification. One of the problems of this approach is that triangles that intersect the octree cells used to partition the model cannot be simplified before the cells are combined in a higher level of the hierarchy. Therefore, the number of triangles in an octree cell may exceed the available main memory.

Another problem of many out-of-core simplification methods is that the geometric distance between the original and simplified models cannot be truly controlled, since the original model does not fit into main memory.

Using our high-quality simplification technique, we implemented an end-to-end out-of-core mesh simplification algorithm that is capable to guarantee a given geometric error between the original and simplified models.

Our method consists of three parts: memory-insensitive cutting, hierarchical simplification and memory-insensitive stitching of adjacent parts. Since the

first and last parts of the algorithm work entirely on disk, and the number of vertices during each simplification step is bound by a constant value, the whole algorithm can process models that are far too large to fit into main memory.

The use of generalized pair contractions combined with cutting of the model at octree cell boundaries allows us to not accumulate the triangles that intersect the octree cells. Also, in contrast to most previous out-of-core simplification approaches, we do not use vertex clustering, since for a given error tolerance the reduction rates are low compared to vertex contraction techniques.

Since we use our high-quality simplification method during the whole reduction, and we guarantee a maximum geometric error between the original and simplified models, the computation time is higher compared to recent approaches, but the gain in quality and/or reduction rate is significant.

1.2.2.3 Intersection-free simplification

All of the existing mesh simplification approaches focus on the creation of a geometrically close approximation of the original model. But none of the standard techniques tries to avoid self-intersections during the simplification process.

When producing a simplified version of a model with close layers, such as dressed humans, self-intersections can result in intolerable results. Even methods that allow the sewing of close surface parts lead to unpleasant artefacts. We show that in real-world situations self-intersections often lead to unacceptable results.

In the final method presented in this thesis, we focus on the avoidance of self-intersections in the case of vertex contraction simplification. This is done by parameterizing the contraction operations over time and by detecting collisions of affected simplices. For the case of a collision, we examined different strategies to determine new target positions that avoid the collision.

Experimental results show that our method produces high-quality simplified meshes without causing any new self-intersections. Furthermore, our approach allows for arbitrary changes in the topology.

1.2.3 Thesis structure

This thesis consists of two main parts:

- **Mesh repair**, in which we present our methods to perform *consistent orientation of normals* (chapter 3) and *progressive gap closing* (chapter 4), and
- **Mesh simplification**, in which we present our approaches for *high-quality* (chapter 5), *out-of-core* (chapter 6) and *intersection-free* (chapter 7) simplification.

But first, in the next chapter, we will present some basic concepts about polygonal meshes and areas of their application, provide an overview of data structures and file formats, which are most commonly used to store manifold and non-manifold meshes, and describe the implementation of data structure used in our work.

Chapter 2

Basics on meshes and data structures

2.1 Polygonal meshes

In this section we define what a polygonal mesh is and discuss what kinds of meshes there are and for which purposes we need meshes. We begin with the definition of a polygonal mesh and its main properties in section 2.1.1. Then, in section 2.1.2, we describe the most important relations between the number of different elements of a mesh as well as topological functions to create and edit polygonal meshes.

In section 2.1.3 we concentrate on a particular simple case of polygonal meshes, triangle meshes, which are used very widely in computer graphics. Finally, in section 2.1.4, we describe and motivate the usage of meshes.

2.1.1 Mesh definition

Let us define the notion of a mesh. Most easily this could be done at the example of a triangle mesh, as shown in figure 2.1. The triangle mesh looks very similar to a graph, which is defined as a set of vertices V and a set of edges E, where each edge $e \in E$ is a set of two vertices $e = v_1, v_2$ specifying the end points of the edge. But a triangle mesh is more than a graph. First of all, it also contains a geometric representation of the vertices, i. e. it is, for example, embedded in the three-dimensional Euclidean space – each vertex is mapped to a point and each edge to a line segment or a curved segment connecting the end vertices. In order to distinguish between the geometric representation of a mesh and its graph-like structure, we introduce the terms mesh geometry and mesh connectivity. We define a mesh $\mathscr{M}$ as a pair $(\mathscr{C}, \mathscr{G})$ of its connectivity

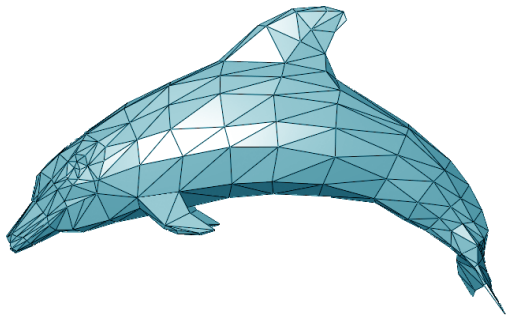


Figure 2.1: Example of a triangle mesh representing a dolphin.

$\mathscr{C}$, which describes the mesh constitutes and their relations to each other, and its geometry $\mathscr{G}$, which describes embedding of the mesh in, for example, the Euclidean space. In what follows we describe separately the mesh connectivity and geometry together with their most important properties.

2.1.1.1 Connectivity

Even without the geometry, the polygonal mesh connectivity differs from a graph, as the triangles itself are represented. A polygonal mesh can contain in addition to triangles also closed polygons as faces. The connectivity of a polygonal mesh contains a third set F of faces. Each face $f \in F$ is a closed loop of edges $f = (e_1, e_2, \ldots, e_n)$. In figure 2.1 the faces are illustrated by shading them opaquely. We can gather the said in the following definition.

Definition 2.1 (polygonal connectivity)

- *The* polygonal connectivity *is a triple* (V, E, F) *of the set of vertices* V*, the set of edges* E *and the set of faces* F*.*
- *Each edge is a subset of* V *with two elements.*
- *Each faces is an* ordered closed loop *of edges* $(e_1, e_2, \ldots, e_n)$ *with* $e_i \in E$*, such that* $e_1 = \{v1, v2\}, \forall i = 2 \ldots n-1 : e_i = \{v_i, v_{i+1}\}$ *and* $e_n = \{v_n, v_1\}$*.*

Here, we want to prevent the reader from the idea that a polygonal connectivity has anything to do with polygons. The connectivity only defines the relations between the vertices, edges and faces. The geometric representation of these can be arbitrarily curved and bent.

The collection of all vertices, all edges and all faces of a mesh is called the *mesh elements*. We next define the relations of *incidence* and *adjacency*.

Definition 2.2 (incidence)

- *A vertex $v \in V$ is* incident *to an edge $e \in E$, iff $v \in e = \{v_1, v_2\}$.*
- *An edge $e \in E$ is* incident *to a face $f \in F$, iff e is contained in the closed loop of f.*
- *The* incidence *relation is reflexive and transitive.*

The reflexive nature of the incidence relation defines the incident edges to a vertex and the incident faces to an edge. Thus it makes sense to talk of the incidence between an edge and a face. The transitive nature of the incidence relation defines the incidence relation between vertices and faces.

Definition 2.3 (adjacency)

- *Two faces are* adjacent*, iff there exists an edge incident to both of them.*
- *Two edges are* adjacent*, iff there exists a vertex incident to both.*
- *Two vertices are* adjacent*, iff there exists an edge incident to both.*

Up to now we defined only terms for very local properties among the mesh elements. Now we move on to global properties.

Definition 2.4 (edge-connectedness)

A polygonal connectivity is edge-connected*, iff each two faces are connected by a path of faces such that two successive faces in the path are adjacent.*

Definition 2.5 (closeness)

A polygonal connectivity is closed*, iff each edge is incident to exactly two faces.*

General polygonal meshes can be very nested. Therefore, we define next the notion of a manifold mesh and a manifold mesh with border.

Definition 2.6 (manifoldness)

A polygonal connectivity is manifold*, iff*

(a) each edge is incident to exactly two faces, and

(b) the non-empty set of faces around each vertex form a closed edge-connected loop.

Definition 2.7 (manifoldness with border)

A polygonal connectivity is manifold with border*, iff*

(a) each edge is incident to one or two faces, and

(b) the non-empty set of faces around each vertex form an open or closed edge-connected loop.

For the navigation in a mesh the orientation of a face, i.e. the order of its edges, is important, especially the consistent orientation of adjacent faces.

Definition 2.8 (orientability)

- *The* oriented half-edge *of an edge e_i in a face $f = (\ldots, \{v_{i-1}, v_i\}, e_i = \{v_i, v_{i+1}\}, \ldots)$ is the pair (v_i, v_{i+1}).*
- *Two adjacent faces are* consistently oriented*, iff the oriented half-edges of their common edge have opposite orientation.*
- *A manifold mesh is* orientable*, iff all adjacent faces are oriented consistently.*

The orientation of a face in a polygonal mesh can be used to define the *outside* of a closed mesh or to calculate the surface normal. It is also important during the navigation through the mesh. The problem with non-orientable meshes is that one cannot choose the orientation of the faces consistently. Therefore, surface normals can not be calculated consistently and no *inside* or *outside* relation makes sense. Furthermore, it complicates the navigation in the mesh, as one must know during the traversal between two adjacent faces, whether the orientation of the face changes or not.

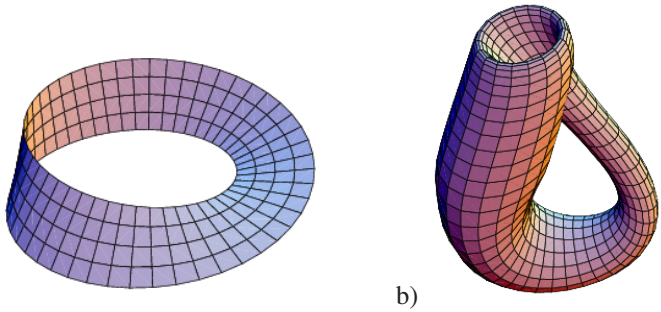
Figure 2.2 shows two classical examples of non-orientable surfaces: Möbius strip and Klein bottle. One can easily check their non-orientability, as one can move on the surface from one point always staying on the same side of the surface in a loop and arrive back at the same point but on the other side of the surface.

Definition 2.9 (polyhedron)

A polygonal mesh is called polyhedron*, iff*

(a) the mesh is closed and edge-connected, and

(b) each vertex $v \in V$ is incident to a finite, cyclic ordered set of faces F_i, i.e. there exists an ordering of faces F_i incident to a vertex v, such that F_i and F_j share an edge incident to v.



a) b)

Figure 2.2: Two classical examples of non-orientable surfaces: a) Möbius strip; b) Klein bottle.

Based on definition 2.9, we can define the polyhedral connectivity as a quadruple (V,E,F,P) of vertices, edges, faces and polyhedra. Each polyhedron is a set of oriented faces forming a topological polyhedron. The local and global relations *incidence*, *adjacency*, *face-connectedness*, *closeness*, *manifoldness* and *manifoldness with border* are direct generalizations of the corresponding attributes in a polygonal connectivity. We do not want to define all these terms in detail, but want to mention that the roll of the face orientation is taken by the outside relation of the topological polyhedron.

2.1.1.2 Geometry

Now we will add the geometry to the mesh. We want to describe this procedure only for the typical case of polygonal geometry in Euclidean space. Similarly, meshes with curved edges and surfaces could be defined.

Definition 2.10 (Euclidean polygonal geometry)

The Euclidean geometry $\mathscr{G}$ *of a polygonal mesh* $\mathscr{M} = (\mathscr{C},\mathscr{G})$ *is a mapping from the mesh elements in* $\mathscr{C}$ *to* $\mathbb{R}^3$ *with the following properties:*

- *a vertex is mapped to a point in* $\mathbb{R}^3$*;*
- *an edge is mapped to the line segment connecting the points of its incident vertices;*
- *a face is mapped to the inside of the polygon formed by the line segments of the incident edges.*

Here, arises a problem that also often arises in practice. In $\mathbb{R}^3$ the edges of a face often do not lay in a plane. Therefore, the geometric representation of a

face is not defined properly, and also a two-dimensional parametrization of the polygon is not easily defined. In practice, this is often ignored and the polygon is split into triangles for which a unique plane is given in Euclidean space.

Often, further attributes like physical properties of the described surface, the surface colour, the surface normal or a parametrization of the surface are necessary. These attributes are typically stored as constant values at the vertices and interpolated along the edges and faces. Otherwise, higher order interpolation schemes are exploited by further attribute values given at the edges and or faces.

2.1.2 Mesh relations and operations

The *Euler* and *Euler-Poincaré* formulas describe the relationship of the number of vertices, the number of edges and the number of faces in a polygonal mesh.

2.1.2.1 Euler formula

For a polygonal connectivity $\mathscr{C} = (V, E, F)$ with $v = |V|$ vertices, $e = |E|$ edges and $f = |F|$ faces, the following *Euler equation* holds (for more details see e.g. the works by Wilson [105] or by Foley et al. [27]):

$$v - e + f = \chi, \tag{2.1}$$

where χ is the *Euler characteristic*. For a closed manifold connectivity, the Euler characteristic splits into the number of *shells* s and the *genus* g of the mesh:

$$v - e + f = 2(s - g). \tag{2.2}$$

A *shell* is an internal void of a solid. A shell is bounded by a manifold surface, which can have its own genus value. Note that the solid itself is counted as a shell. Therefore, the value for s is at least 1.

The *genus* of a closed surface is the number of handles of the described solid, i.e the number of holes that it. The surface of a cup has, for example, one handle, i.e. one hole in the circumscribed solid.

As we will deal with triangular meshes, we can also consider the special characteristic of triangular meshes to derive a much simpler equation. For this, we enumerate in a closed manifold triangle mesh all incidences between edges and triangles. In terms of edges, there are $2e$ incidences as each edge is incident to two faces. In terms of triangles, there are $3f$ incidences resulting in

$$2e = 3f. \tag{2.3}$$

Substitution of this relation in the Euler equation yields

$$2v - f = 4(s - g), \tag{2.4}$$

and for edge-connected triangle meshes with genus one, we get

$$f = 2v. \tag{2.5}$$

Although the number of conditions to this relation is large, it is a good approximate statement for meshes, that describe the surface of a solid. The Euler characteristic χ is typically small compared to the number of vertices and triangles. For completeness, we want to incorporate the number of border edges b into the equations. As each border edge has only one incident face, the number of incidences between edges and faces in terms of the edges must be corrected to $2e - b$ resulting in

$$2v - f - b = 4(s - g). \tag{2.6}$$

2.1.2.2 Euler-Poincaré formula

Let r be the number of face inner loops. Then the generalized *Euler-Poincaré formula* is the following:

$$v - e + f - r = 2(s - g). \tag{2.7}$$

And with $r = l - f$, where l is the number of all outer and inner loops of faces:

$$v - e + f - (l - f) - 2(s - g) = 0. \tag{2.8}$$

The Euler-Poincaré formula describes the topological property amount vertices, edges, faces, loops, shells and genus. Any topological transformation applied to the model will not alter this relationship.

2.1.2.3 Euler operators

Once a polyhedron model is available, one might want to edit it by adding or deleting vertices, edges and faces to create a new polyhedron. These operations are called *Euler operators* (for more details on modelling using Euler operators see e.g. the works by Eastman and Weiler [24] or by Mäntylä and Sulonen [67]). However, it has been shown that in the process of editing a polyhedron with Euler operators, some intermediate results may not be valid solids at all.

Based on the the relation 2.8, some Euler operators have been selected for editing a polyhedron so that the Euler-Poincaré formula is always satisfied. There are two groups of such operators: the *Make* group and the *Kill* group. Operators starting with *M* and *K* are operators of the Make and Kill groups, respectively.

Euler operators are written as *Mxyz* and *Kxyz* for operations in the Make and Kill groups, respectively, where x, y and z are elements of the model (e.g.

Operator	Meaning	v	e	f	l	s	g
MEV	Make an edge and a vertex	$+1$	$+1$				
MFE	Make a face and an edge		$+1$	$+1$	$+1$		
$MSFV$	Make a shell, a face and a vertex	$+1$		$+1$	$+1$	$+1$	
MSG	Make a shell and a hole					$+1$	$+1$
$MEKL$	Make an edge and kill a loop		$+1$		-1		

Table 2.1: The Make group of Euler Operators.

Operator	Meaning	v	e	f	l	s	g
KEV	Kill an edge and a vertex	-1	-1				
KFE	Kill a face and an edge		-1	-1	-1		
$KSFV$	Kill a shell, a face and a vertex	-1		-1	-1	-1	
KSG	Kill a shell and a hole					-1	-1
$KEML$	Kill an edge and make a loop		-1		$+1$		

Table 2.2: The Kill group of Euler Operators.

vertex, edge, face, loop, shell and genus). For example, MEV means adding an edge and a vertex while KEV means deleting an edge and a vertex.

Note that not all operators that can be represented in this way are Euler operators: MVF is not an Euler operator, since it adds a vertex and a face to a model, and therefore, violates the Euler-Poincaré formula.

It has been proved by Mäntylä [65] that Euler operators form a complete set of modelling primitives for manifold solids. More precisely, every topologically valid polyhedron can be constructed from an initial polyhedron by a finite sequence of Euler operators. Therefore, Euler operators are powerful operations.

The Make group consists of four operators for adding some elements into the existing model creating a new one and a *Make-Kill* operator for adding and deleting some elements at the same time.

Table 2.1 shows the change of values of v, e, f, l, s and g. Note that adding a face produces a loop, the outer loop of that face. Therefore, when f is increased, l should also be increased. This new loop and the new face will cancel each other in the subexpression $l - f$. None of these operators would cause the Euler-Poincaré formula to fail.

The Kill group just performs the opposite of what the Make group does. In fact, replacing the M and K in all Make operators with K and M, respectively, would get the operators of the Kill group. Five operators of the Kill group are shown in table 2.2.

2.1.3 Triangular meshes

A lot of algorithms that deal with meshes are restricted to the simple case of triangular meshes. This is easily justified from the much simpler handling of triangles in terms of intersection calculation, attribute interpolation, line up of physical equations, rendering and so on. But in real world data, a lot of meshes contain not only triangles.

For this reason, a whole area of research has tackled the problem of efficiently subdivide a simple polygon, i.e. a polygon without self-intersections and only edges of non-zero length, into triangles. That this is always possible is intuitively clear from the idea of the basic algorithm for this task. It is called *ear-cutting* and does exactly the following.

The algorithm searches three successive vertices on the polygon forming a corner also called *ear*). For each ear, it checks, whether the line segment from the first to the third vertex is completely contained in the polygon, or if it is in the outside or intersecting the polygon. If the line segment is inside the polygon, the triangle formed by the ear can be cut away, and this process is repeated until only one triangle is left. The cut-away triangles form the polygon's triangulation. The intuition tells us that there cannot be a polygon, from which we cannot cut at least one ear.

The expensive operation is the intersection test of the line segment and the polygon. Chazelle [16] came up with a solution to the polygon triangulation problem that can be computed in linear time in the number of edges in the polygon. Two other papers, one by Kirkpatrick et al. [54] and another one by Seidel [87], give simpler solutions with only slightly worse running time. Held [41] developed a very robust implementation, that always succeeds also on polygons with self-intersections and degenerated edges.

Without loss of generality, we can assume that the model consists entirely of triangular faces, since any non-triangular polygons may be triangulated in a pre-processing phase [72, 87].

It's possible that a model contains isolated vertices and edges, which are not part of any triangle. For best results in practice, we should maintain them during simplification and render them at run-time [62, 79, 84, 85]. However, to streamline the discussion, we will assume that models consist of triangles only. For most algorithms, the only effect of isolated vertices and edges is to complicate the implementation; the underlying algorithms remain the same. Finally, we will also assume that the connectivity of the model is consistent with its geometry, i.e. if the corners of two triangles coincide in space then those triangles share a common vertex.

2.1.4 Application areas for meshes

There are several important application areas for meshes. One of the most important ones is in *finite element simulations*. Here, a surface is split into a polygonal mesh and attributed with physical quantities of the underlying material. The equations of motion are written in terms of the mesh elements, and equation solvers are used to find solutions for different starting conditions. The flexible structure of a mesh allows to model arbitrary geometries.

The *finite element method (FEM)* has been successfully applied to simulate all types of materials including fluids and cloth. Therefore, the FEM is widely used in all industrial branches. One common task in FEM is the generation of appropriate meshes from boundary data only. The mesh elements of the produced meshes must fulfill certain quality criteria. More information on this topic can be found in a book by George [32].

The second main application of meshes is the *boundary representation of objects*. Here, polygonal and triangular meshes come into operation. The meshes are typically attributed with the surface normal, surface material information and a parameterization together with some textures specifying fine variations of the surface colour, surface normal or of the surface offset in direction of the surface normal. Simple triangle meshes are very common because of their hardware accelerated rendering with all the mentioned attributes.

The boundary representation of objects is used, for example, in *computer aided design (CAD)*, in *virtual worlds*, in the *game industry*, for *terrain modelling*, and gains more and more importance in *electronic commerce*. New objects are often scanned with 3D scanners producing very fine and large meshes, which demand for simplification.

Scientific visualization is also a broad application area for meshes. Not only the finite element meshes are directly visualized, but new surface meshes are generated to represent and visualize isosurfaces in volume data sets.

Finally, meshes are also used as algorithmic tool for *spatial hashing* and to build hierarchical structures for *point location queries*.

2.2 Data structures and file formats for meshes

In this section we discuss different data structure for triangle meshes, both manifold (in section 2.2.1) and non-manifold (in section 2.2.2), and then introduce the one that we used in our work. At the end, in section 2.2.4, we briefly describe the most common file formats used in computer graphics to store polygonal meshes.

A boundary representation of a polyhedral surface consists of a set of vertices V, a set of edges E and a set of faces F. For navigation in the mesh the incidence and the adjacency relations (see definitions 2.2 and 2.3 from sec-

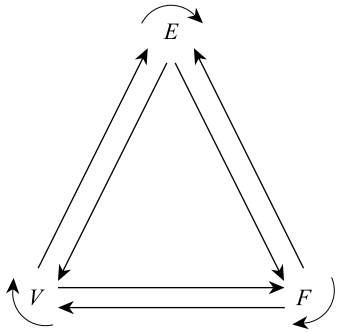


Figure 2.3: Possible adjacency relations in a boundary data structure.

tion 2.1.1.1) discussed by Weiler [98] are used (this is discussed in detail in, e.g., works by Mäntylä [66] or by Hoffmann [43]).

Figure 2.3, which we adopted from Woo [106], shows all possible adjacency and incidence relations in a data structure based on three primitive topological entities (face F, edge E and vertex V). The arrows between the sets of different mesh elements represent the incidence relations, whereas the self-pointing arrows illustrate the adjacency relations. For example, the relation $E \rightarrow V$ stores two incident vertices for each edge, and the relation $V \rightarrow V$ stores all adjacent vertices for each vertex.

In order to answer all possible incidence and adjacency questions, one needs to represent only a subset of all relations. This subset must connect all sets of equal and different mesh elements, i.e. there need not only be a path from each mesh element set to each other, but there also must be a path from each mesh element set back to the set in order to answer adjacency questions. In figure 2.3, this means that one can eliminate arrows as long as this condition is fulfilled. For example, the relations $V \rightarrow E \rightarrow F \rightarrow V$ would suffice.

It is often possible to represent some of the relations only partially by one or two representatives. For example, if the relations $F \rightarrow F \rightarrow V$ are known in a manifold mesh with border, the relation $V \rightarrow F$ can be stored with one face per vertex, as the remaining faces of each vertex can be determined through the adjacency relation of the faces. To indicate this, we write near the respective arrow the number of representatives stored per relation (see, for example, figure 2.4).

It is clear that the enumeration of all faces incident on a vertex is more expensive than if the relation would have been stored explicitly. On the other hand the update of the explicit representation is more expensive. Ni and Bloor [74] provide a good analysis of different possible data structures.

Of course, the more explicit information is stored, the more memory is required, but the faster is the access to the required data. Unfortunately, memory is a hard limit in computer systems. Even virtual memory does not help in most cases, since the mapping of a complex triangle mesh to the linear structure of the memory usually opposes the need for local coherence access to memory. Drastic slowdown of the performance due to extensive swapping is the result.

Performance is a very important aspect, but time is in general a soft limit. When doubling the amount of input data to be processed, it is acceptable to wait even more than twice the time to get a result, if this is necessary. But if doubling the amount of data implies that the data structure does not fit into the memory of a computer, this makes it impossible to work on a data set.

As a result, the implementation of a data structure can be considered as a compromise between memory requirements and performance gains.

The following short survey of edge-based data structures addresses their sufficiency for modelling topology and the efficiency of their primitive operations and storage costs.

According to definition 2.6 from section 2.1.1.1, the two types of boundary representations are *manifold* and *non-manifold* surfaces. A manifold surface is a surface, where for each point on the surface there exists a neighbourhood that is homeomorphic to the open disc. Non-manifold examples are two tetrahedrons glued together at a single vertex or a common edge.

2.2.1 Manifold mesh data structures

Several manifold data structures have been developed to provide fast access (ideally $\mathcal{O}(1)$) to the information that is required by different algorithms. The most popular ones are the *winged-edge* [8], *half-edge* [99] and *quad-edge* [36] data structures. The half-edge data structure suggested by Mäntylä [66] turned out to be very suitable for polygonal meshes (see Kettner [53]).

2.2.1.1 Winged-edge data structure

Perhaps the oldest data structure for a boundary representation is Baumgart's *winged-edge* structure [8, 9, 33]. For each oriented edge it stores eight references: two vertices (PVT, NVT), two faces ($PFACE$, $NFACE$) and four incident edges that share the same faces and vertices (PCW, $PCCW$, NCW and $NCCW$), the so-called *wings* (see figure 2.5). An edge is oriented from the source vertex PVT to the target vertex NVT. The face $PFACE$ is to the left of the oriented edge when the surface is seen from the outside. Vertices and faces have a single pointer to one incident edge.

This data structure is able to model orientable manifolds (see definition 2.8 from section 2.1.1.1). It is even sufficient for curved-surface environments where loops and multi-edges are allowed, as shown by Weiler [99]. The basic

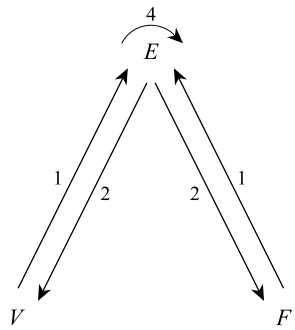


Figure 2.4: Adjacency relations in the winged-edge data structure.

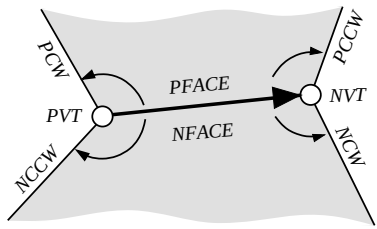


Figure 2.5: Winged-edge data structure.

operations include traversal around a vertex and around a face. High-level operations maintaining integrity are Euler operators, which we discussed in section 2.1.2.3. The next edge counterclockwise around a vertex V for an edge E is equal to $E{\rightarrow}PCW$ if $E{\rightarrow}PVT$ is equal to V and $E{\rightarrow}NCW$ otherwise.

Variants are possible where vertex and face pointers can even be omitted without loosing the traversal capabilities knowing the edge visited previously. However, all four edge pointers must remain if loops or multi-edges are allowed since otherwise the traversal around a vertex or face is no longer uniquely defined (see Weiler [99]). The winged-edge data structure where the wings *PCCW* and *NCCW* are omitted has been called *Doubly Connected Edge List* (DCEL) (see Muller and Preparata [69]), though this name is now more commonly used for the half-edge data structure (see de Berg et al. [21]).

The two symmetric parts in the winged-edge correspond to the two possible orientations of the edge. The inefficient case distinction in the traversal computation results from the fact that a pointer to an edge does not encode the orientation it is currently used with. One extension of the winged-edge maintains an additional bit with each edge-pointer to code the orientation, but this leads to cumbersome storage layouts and function interfaces.

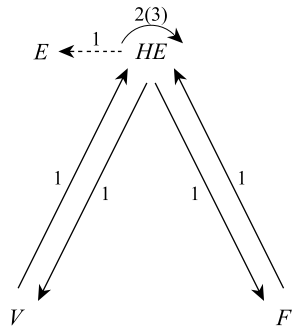


Figure 2.6: Adjacency relations in the half-edge data structure.

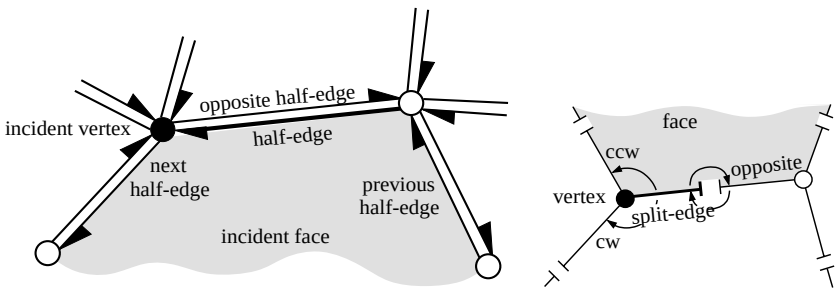


Figure 2.7: Half-edge and split-edge data structures.

The winged-edge data structure has been termed "edge-based" because its three major stored relations have edge as the reference entity. Weiler [99] noticed that representing the edge as a single structure complicates relation-query algorithms because it requires additional processing to determine which side or end of the edge is intended every time a reference to an edge structure appears. He defined three improvements. The first modifies the winged-edge data structure by adding explicit *edge-side* indicators. Two other data structures – the *vertex-edge* and the *face-edge* – split each edge into two *half-edges*. Each half-edge is related to one of the two edge ends or sides and is also associated with its opposite half-edge.

2.2.1.2 Half-edge data structure

As mentioned above, the orientation problem can be solved for the winged-edge data structure by splitting the edge into the two symmetric records called *half-edges* and adding mutual links to each other. There are two ways of split-

ting the edge, which are actually dual to each other: *split-edge* (see Eastman [23]) and *half-edge* (see Weiler [99]). In both situations the half-edge contains a pointer to an incident vertex, an incident face and the opposite half-edge.

The half-edge data structure was first introduced by Mäntylä and Sulonen [67], though in their solid modelling system they used an additional edge record between two opposite half-edges, making this access less efficient.

It is a matter of convention whether the source or target vertex is the one chosen to be stored in a half-edge or whether the face to the left or the right is stored. Weiler [99] chooses the source vertex and the face to the right. The half-edge data structure, shown in figure 2.7 together with the dual split-edge data structure, additionally stores a pointer to the next clockwise half-edge and optionally a pointer to the previous counterclockwise half-edge around the face. It is, therefore, biased towards traversals around the incident face. Its next and optional previous pointer refer to half-edges counterclockwise and clockwise around the incident vertex. The traversal operation that is not directly accessible with a single pointer access is available through the opposite half-edge. For example, the next half-edge around the incident source vertex is $opposite() \rightarrow next()$. The different conventions are not independent. If the convention defines the half-edge order around a face to be clockwise, the half-edge order around the vertex will be counterclockwise, and vice versa.

The half-edge data structure is able to model orientable manifolds. It is sufficient for modelling topology even in the presence of loops and multi-edges, which can occur in curved-surface environments. High-level operations maintaining integrity are again Euler operators (see section 2.1.2.3).

2.2.1.3 Quad-edge data structure

If we perform both halving steps for the half-edge data structure, we end up with the *quad-edge* data structure, suggested by Guibas and Stolfi [36]. It provides a fully symmetric view on the primal and the dual graph, as can be seen in figure 2.9. Instead of using opposite pointers, a two-bit counter r is used to address a slot in an edge record of four quad-edges. With an additional bit f per edge for the flipped status the quad-edge data structure is able to model non-orientable manifolds.

A quad-edge data structure is defined as an edge algebra with three operations: $Rot()$, $Flip()$ and $Onext()$. An edge is represented as a triple (e, r, f) with $r \in \{0,1,2,3\}$ and $f \in \{0,1\}$. e is the base pointer to the quad-edge record with the four incident edges $e[0]$ to $e[3]$. The operations are implemented as follows with a calculus modulus 4 for r and modulus 2 for f:

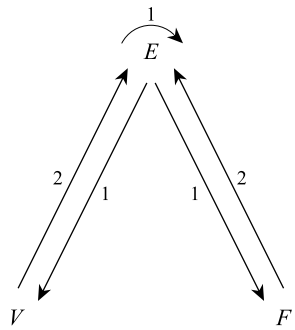


Figure 2.8: Adjacency relations in the quad-edge data structure.

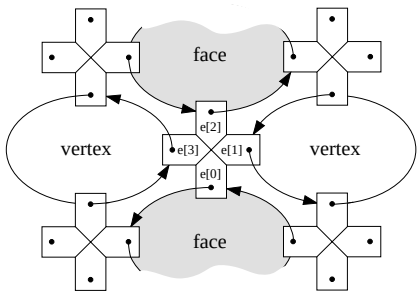


Figure 2.9: Quad-edge data structure.

$$\begin{aligned} Rot(e,r,f) &= (e, r+1+2f, f), \\ Flip(e,r,f) &= (e, r, f+1), \\ Onext(e,r,f) &= Flip^f(Rot^f(e[r+f])). \end{aligned} \tag{2.9}$$

Four different orientations of an edge are considered: two orientations from vertex to vertex and two orientations for the dual edge from face to face. The $Rot()$ operator rotates the edge by 90 degrees, oscillating between the primal and the dual view of the structure. For non-orientable manifolds an edge can additionally be seen from above or below the surface, which is encoded in the f bit. The $Flip()$ operation changes the view from above to below or vice versa. The $Onext()$ operation gives the next quad-edge in counterclockwise order around the source vertex (origin) or the next quad-edge in clockwise order, if f is equal to 1. The values for $Onext()$ are simply stored in the record for each edge (i.e. four pointers and four times three bits for r and f). The

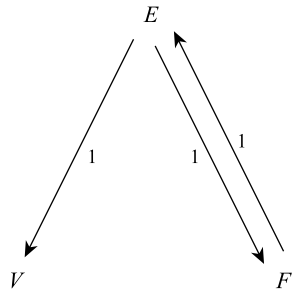


Figure 2.10: Adjacency relations in the face-edge-vertex data structure.

operations simplify considerably for orientable manifolds. They can be further simplified if the dual graph is not necessary. This reduces to the winged-edge data structure enriched with a bit to encode orientation.

The single high-level operation that modifies a quad-edge data structure is the *Splice*() operation. It is its own dual. The usual Euler operators (see section 2.1.2.3) can be implemented in terms of *Splice*(). The quad-edge data structure provides a unified view for the primal and dual graph. This implies that vertices and faces cannot be distinguished with strong type checking at compile time.

The definition used for duality implies, furthermore, that the faces must have a single connected boundary. Holes in faces are not allowed. If strong type checking is desired, the *Splice*() operation is needed twice, once for the primal view and once for the dual view. *Splice*() can also be provided for the half-edge data structure.

2.2.1.4 Face-edge-vertex data structure

Ni [73] designed the *face-edge-vertex* data structure for a study of free-form solid modelling that used a hybrid CSG/B-Rep (constructive solid geometry and boundary representation) approach. The data structure maintains three relations: $E \rightarrow V$, $E \rightarrow F$ and $F \rightarrow E$ (see figure 2.10). The first two relations are returned by a boundary evaluation procedure (see Requicha and Voelcker [81]) on the CSG model, and the third is derived from the previous two. We can create a free-form solid model in three stages by creating, first, a basis CSG model with geometric coverage of only quadric surfaces and, second, free-form surfaces in B-spline form. Then we perform operations between the basis CSG model and the free-form surfaces, building the face-edge-vertex data structure for the basis model and updating the data structure accordingly.

Compared with the winged-edge, the face-edge-vertex data structure shows an important feature: its manipulation algorithms (creation and modification) are much simpler. If a solid model is newly created or modified, the winged-edge must invoke an elaborate and error-prone pointer-chasing procedure for setting $E \rightarrow E$ relations correctly. One edge can refer to the other four edges when and only when the other four are actually created or modified. In the face-edge-vertex data structure, updating can proceed face by face, which is more natural to the user. For this reason, we can say the face-edge-vertex data structure is face-based.

2.2.2 Non-manifold mesh data structures

In practice, many objects represented by triangle meshes contain isolated vertices or edges that are locally non-manifold for several reasons (see Campagna et al. [15]). On the one hand, many data structures are restricted to orientable manifold triangle meshes. Even if there are only some few artefacts that are locally non-manifold within a large data set, these data structures are not able to represent such an object.

On the other hand, several data structures, which are capable of storing non-manifold meshes require more memory due to the fact that they store non-manifold information all over the whole object by using extra memory, even if most entities are in fact manifold.

Several sophisticated data structures have been developed, but most of them are capable to represent general polygonal meshes instead of just the special case of a triangle mesh. Specializing to triangles obviously allows to design more efficient data structures.

2.2.2.1 Directed-edge data structure

In *directed-edge* data structure by Campagna et al. [14], a single triangle is represented by three directed edges (see figure 2.12). Thus, the common edge $v^a v^b$ of two neighbouring triangles corresponds to two directed edges $v^a \rightarrow v^b$ and $v^b \rightarrow v^a$. This is very similar to the concept of half-edges, discussed in section 2.2.1.2. For each directed edge the following information is stored: a starting vertex v^a, a target vertex v^b, the previous, next and neighbouring edges e^{pv}, e^{nx} and e^{ng}.

All directed edge information is stored in an array, such that the i'th triangle is represented by the directed edges at the entries $3i$, $3i+1$ and $3i+2$. Therefore, there is no need to store explicit references from each triangle to an edge and vice versa. Instead, these references are derived from the given algorithmic context.

So far, only orientable manifold triangle meshes are considered. Assuming that the number of non-manifold vertices or edges is typically small compared

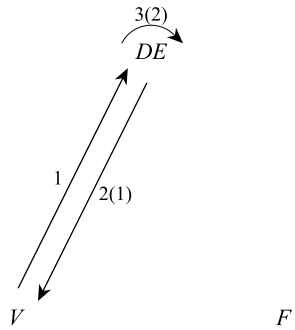


Figure 2.11: Adjacency relations in the directed-edge data structure.

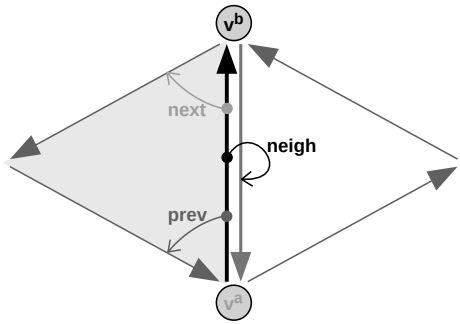


Figure 2.12: Directed-edge data structure.

to the complexity of the whole mesh, the following simple strategy is used to handle non-manifold entities.

Using integer values as array-indices for the references to neighbouring edges allows to use the sign-bit without any additional implementation efforts. The value for the neighbouring edge e^{ng} is zero or positive for every pair of directed edge mates that represent a manifold interior edge. A directed edge on a topological boundary is marked by a certain negative entry for e^{ng}, e.g. -1. A non-manifold edge or an edge whose two triangles are of opposite orientation may be marked by -2.

A manifold vertex references one of its emanating edges. For non-manifold vertices the same strategy can be used as for edges. A negative array-index indicates such a node. Removing the negative sign provides a positive value that points to a separate array which lists either all edges emanating from that vertex or the connected components attached to that vertex.

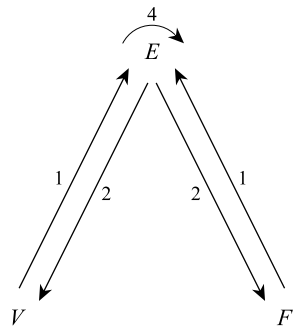


Figure 2.13: Adjacency relations in the radial edge data structure.

2.2.2.2 Radial edge data structure

The *radial edge* data structure is an edge-based boundary representation for non-manifold models presented by Weiler [100, 101, 102]. It is known that there are three kinds of cyclic ordering in non-manifold topology: loop, radial and disk cycles. Note that the loop cycle is a cycle of edges on the boundary of a face, the radial cycle is a cycle of faces incident to an edge, and the disk cycle is a cycle of edges incident to a vertex. The radial edge data structure extends the manifold representation to the non-manifold representation through a radial cycle.

Weiler's radial edge data structure is a generalization of Baumgart's winged-edge data structure, disscussed in section 2.2.1.1, to non-manifold geometry. These two data structures allow us to answer questions about topological adjacency relationships often either in constant time or in time proportional to the size of the output set. Weiler's data structure also records the radial ordering of faces around non-manifold edges, hence its name (see figure 2.14). Radial edge structure was conceived for non-manifold modelling and Weiler has proven its completeness, which means that any adjacency relationship can be extracted from this representation.

In order to describe the topology of a spatial subdivision, the radial edge representation employs the concept of *use* of a topological element. A use can be seen as the occurrence of a topological element in an adjacency relationship related to an element of higher dimension. Thus, the radial edge structure explicitly stores the two uses (sides) of a face by the two regions (not necessarily distinct) that share that face. Each face-use is bounded by one or more loop-uses, which are composed by an alternating sequence of edge-uses and vertex-uses (see figure 2.14). Vertex-uses are necessary to store non-manifold conditions at vertices.

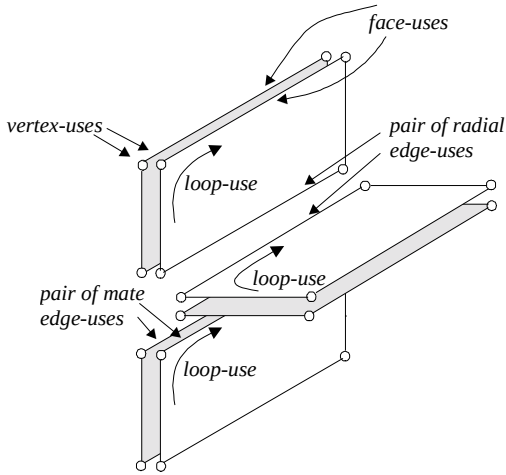


Figure 2.14: Radial edge data structure: uses of topological elements.

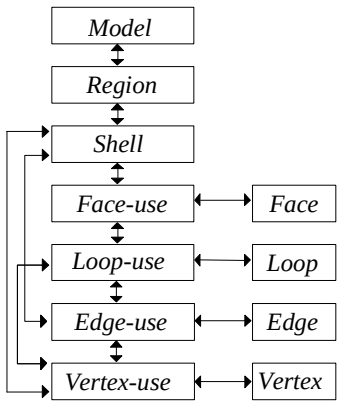


Figure 2.15: Radial edge data structure: hierarchy of topological entities.

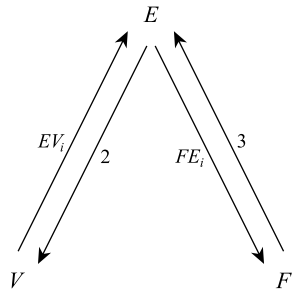


Figure 2.16: Adjacency relations in the progressive simplicial complexes data structure containing simplices of dimension 0, 1 and 2. EV_i and FE_i represent the number of edges incident to a vertex and the number of faces incident to an edge, respectively.

The radial edge structure is a hierarchical description of a spatial subdivision, starting in higher dimension levels (regions) and reaching the lower levels (vertices) (see figure 2.15). The topological elements are kept in doubly linked circular lists and have pointers to their attributes.

Topological data structures are complex and should not be directly manipulated. Weiler has introduced a set of operators that provide a high-level method to access the radial edge structure. These operators are divided in two groups. The first group has operators that act on faces of a spatial subdivision and are analogous to the (manifold) operators presented by Mäntylä [66]. The second group has operators that are capable of creating wireframes and adding faces, which are attached to specified edges or wireframes. They are referred to as non-manifold operators. Considerations about a minimal set of operators can be found in the paper by Wu [108].

2.2.2.3 Progressive simplicial complexes

Progressive simplicial complexes (PSC) is a data structure developed by Popović and Hoppe [79] for storing and transmitting triangular models. It is non-oriented, i.e. it is stripped off any oriented simplices or cells. Consequently, progressive simplicial complexes can be used to represent both orientable and non-orientable objects (see definition 2.8). Thus, unlike oriented boundary representations (e.g. *directed-edge* data structure discussed in section 2.2.2.1), there is no simplex redundancy. Besides, PSC data structure is able to represent n-dimensional simplicial complexes. However, the lack of explicit oriented simplices or a geometric orientation mechanism for simplices in this data

structure poses some difficulties in rendering. Unlike *progressive meshes* developed earlier by Hoppe [44], progressive simplicial complexes avoid explicitly storing surface normals at vertices. Instead, this data structure makes usage of smoothing group fields for different materials.

In PSC representation, the geometry of a triangular model is denoted as a tuple (K,V), where the *abstract simplicial complex* K is a combinatorial structure, which specifies the adjacency of vertices, edges, triangles, etc., and V is a set of vertex positions specifying the shape of the model in $\mathbb{R}^3$.

An abstract simplicial complex K consists of a set of vertices together with a set of non-empty subsets of the vertices, which are called the *simplices* of K, such that any set consisting of exactly one vertex is a simplex in K, and every non-empty subset of a simplex in K is also a simplex in K.

The abstract simplicial complex K is not restricted to manifolds (see definitions 2.5 and 2.6, but may in fact be arbitrary. To represent K in memory, the incidence graph of the simplices is encoded using the following linked structures:

```
struct Simplex
{
    int dim;       // 0=vertex, 1=edge, 2=triangle, ...
    int id;
    Simplex* children[MAXDIM+1];       // [0..dim]
    List<Simplex*> parents;
};
```

2.2.3 Our data structure

For progressive gap closing and high-quality simplification techniques, which will be discussed in chapters 4 and 5, respectively, we need an abstraction of different topological entities. As such abstraction we use a *simplex*, which in our data structure represents one of the three topological types, a vertex, an edge or a face – a concept similar to the progressive simplicial complexes data structure (see section 2.2.2.3). The *simplex* basic type allows us to efficiently build and handle contraction pairs used in our *generalized pair contractions* simplification method (see section 5.1).

Inside the *simplex* type we declare all operations and queries, which are common for all topological entities, e.g. “is boundary?”, “is manifold?” “get neighbourhood”, etc. Depending on the actual type of the simplex, a vertex, an edge or a face, it contains references to respective incident or adjacent simplices.

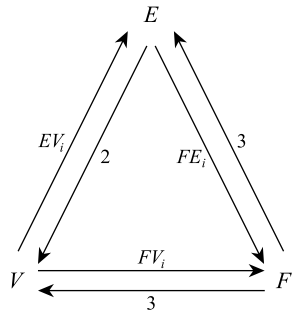


Figure 2.17: Adjacency relations in our data structure. EV_i, FV_i and FE_i represent the number of edges incident to a vertex, the number of faces incident to a vertex and the number of faces incident to an edge, respectively.

2.2.3.1 Memory consumption

As figure 2.17 shows, in addition to the relations stored in the progressive simplicial complexes data structure, we store another two incidence relations: $V \rightarrow F$ and $F \rightarrow V$. This makes our data structure more memory-consuming. However, in several operations essential for the methods presented in this thesis, such as calculation of error quadrics, these relations are used very frequently.

It would suffice to access the information provided by these relations indirectly, i.e., by using relations $V \rightarrow E \rightarrow F$ and $F \rightarrow E \rightarrow V$, but in that case, more additional operations would be necessary, e.g., we would have to check if a respective edge is part of a boundary etc. Since we store the relations $V \rightarrow F$ and $F \rightarrow V$, we are provided with the necessary information directly and, therefore, can perform the important operations such as calculation of error quadrics with the maximum possible speed. This allows us to increase the overall performance of the respective methods.

2.2.3.2 Handling mesh attributes

The most important attribute of a mesh is the geometric representation of the vertices. In nearly all applications the geometry of a vertex is stored as a two-, three- or four-dimensional point in the Euclidean space. The number format depends on the application, but floating point values are most commonly used. The geometric representation of the edges and faces is often not stored explicitly but is derived from the points of the vertices. The edges are mapped to the line segments between their end points and the faces to the polygon interiors described by the line segments of their edges.

Further attributes, such as surface normals, surface colours and texture coordinates, i.e. the surface parametrization, are often available at the vertices. The *vertex attributes* can simply be stored by extending the dimension of the point in order to include the normal, colour and texture coordinates. It is also not a problem to treat attributes given at the faces (or edges, which occur seldom in praxis), as their indices are known.

Often it is necessary to handle attributes at the corner of faces – also called *corner attributes* – if the represented surface has non-differentiable creases. At vertices on the creases the surface normal is not continuous, and different normals might have to be stored for each face the vertex is incident to. The corner attributes can be stored within triangle faces – three per face, in the same order as vertices. The edge attributes in the case when the surface has non-differentiable creases can be handled similarly.

2.2.4 File formats

Anyone who has worked in the field of computer graphics for even a short time knows about the bewildering array of storage formats for graphical objects. It seems as though every programmer creates a new file format for nearly every new programming project. However we will not follow this "strategy" and will use some of the most popular 3D graphics formats. Fortunately, there is a couple of tools available (e.g. Deep Exploration developed by Right Hemisphere[1]), which can convert from one format to another, and therefore, allow us to work with 3D meshes stored in almost any available format.

There is a great variety of different 3D graphics file formats around: from the very basic ones, such as Stanford Triangle Format (PLY) or Stereolithography (STL), which represent just an indexed face set or a triangle soup respectively, to such advanced ones, such as Open Inventor (IV) or VRML (WRL), which allow to represent anything from a single object to a complete scene with interaction as a tree structure called the *scene graph*.

However our applications work only with meshes, and therefore, we are only interested in the way how the formats we are going to read store the mesh information. In the following, we will describe and discuss the formats supported by our applications directly.

2.2.4.1 PLY file format

The PLY polygon file format developed by Turk [94], also known as the Stanford Triangle Format, is a simple object description in form of a single *indexed face set* that was developed at the Stanford University. The PLY file format has

[1] http://www.righthemisphere.com

two variants: an ASCII representation and a binary version. PLY files have the *.ply* extension.

The PLY format describes an object as a collection of vertices, faces and other elements, along with properties such as colour and normal direction that can be attached to these elements. A PLY file contains the description of exactly one object. Sources of such objects include: hand-digitized objects, polygon objects from modelling programs, range data, triangles from marching cubes (isosurfaces from volume data), terrain data, radiosity models. Properties that might be stored with the object include: colour, surface normals, texture coordinates, transparency, range data confidence and different properties for the front and back of a polygon.

A PLY file (see listing consists of a header followed by a list of vertices and then a list of polygons. The header specifies how many vertices and polygons are in the file and also states what properties are associated with each vertex, such as coordinates, normals and colour. The polygon faces are simply lists of indices into the vertex list, and each face begins with a count of the number of elements in each list.

A typical PLY object definition is simply a list of (x, y, z) triples for vertices and a list of faces that are described by indices into the list of vertices. Most PLY files include only this core information.

```
ply
format ascii 1.0   {ascii or binary, format version}
comment this file contains a tetrahedron model
element vertex 4   {there are 4 vertex elements}
property float32 x {vertex contains 3 float coordinates}
property float32 y
property float32 z
element face 4     {there are 4 face elements}
property list uint8 int32 vertex_index
end_header         {delimits the end of the header}
1 0 0              {start of vertex list}
0 1 0
0 0 1
0 0 0
4 1 3 2             {start of face list}
4 3 1 0
4 2 0 1
4 0 2 3
```

Listing 2.1: The model of tetrahedron stored in ASCII PLY file format.

2.2.4.2 Wavefront object file format

Initially, *object* files (see Wavefront format description [83]) were developed to define the geometry and other properties for objects in Wavefront's Advanced Visualizer animation package. In Wavefront's 3D software, geometric object files may be stored in ASCII format (using the *.obj* file extension) or in binary format (using the *.mod* extension). The binary format is proprietary and, according to Murray [71], undocumented; therefore, only ASCII format is supported in our applications and discussed here.

The OBJ file format supports both polygonal objects and free-form objects. Polygonal geometry uses points, lines and faces to define objects, while free-form geometry uses curves and surfaces. Lines and polygons are described in terms of their points, while curves and surfaces are defined with control points and other information depending on the type of curve. The format supports rational and non-rational curves, including those based on Bezier, B-spline, Cardinal (Catmull-Rom splines) and Taylor equations.

OBJ files do not contain colour definitions for faces, although they can reference materials that are stored in a separate material library file. The most commonly encountered OBJ files contain only polygonal faces.

```
g tetrahedron      {start of the tetrahedron group}
v 1 0 0            {start of vertex list}
v 0 1 0
v 0 0 1
v 0 0 0
# 4 vertices
f 2 4 3            {start of face list}
f 4 2 1
f 3 1 2
f 1 3 4
```

Listing 2.2: The model of tetrahedron stored in Wavefront Object file format.

2.2.4.3 Inventor file format

Open Inventor is probably the most widely used C++ graphics toolkit in the world. Originally developed by SGI, Open Inventor has been implemented on almost every major platform. Anything from a single object to a complete scene can be represented as a tree structure called the *scene graph*. The scene graph contains *nodes* representing geometry, transformations, attributes, lights and other data.

IV is the abbreviation for *Inventor*, a file format used by SGI with a variety of programs and now shared with Open Inventor, the successor to Inventor. Inventor files have the *.iv* extension.

The version number indicates the version of the Open Inventor file format, in this case version 2.0. The keywords *ascii* and *binary* indicate whether the rest of the file is in a human-readable ASCII text format or not. The binary files are smaller than the text files, but files in text format can easily be edited manually (see Inventor nodes reference [103]).

```
#Inventor V2.0 ascii

DEF tetrahedron Separator {
    Coordinate3 {
        point [
            1 0 0,
            0 1 0,
            0 0 1,
            0 0 0
        ]
    }
    IndexedFaceSet {
        coordIndex [
            1, 3, 2, -1,
            3, 1, 0, -1,
            2, 0, 1, -1,
            0, 2, 3, -1
        ]
    }
}
```

Listing 2.3: The model of tetrahedron stored in Inventor 2.0 file format.

2.2.4.4 VRML file format

VRML (Virtual Reality Modelling Language, see VRML manual [40]) has become the de facto standard for 3D data on the Internet. Originally conceived as a format to describe multi-user immersive 3D worlds, VRML is finding many more uses as a format for sharing 3D data, for example, engineering models, and for packaging 3D presentations that combine sound and animation. It gained acceptance as both an exchange format and a multimedia content type.

In many respects, Open Inventor is VRML, or at least the parent of VRML. The VRML 1.0 specification was taken directly from Open Inventor's file format, with the addition of a few nodes specific for World Wide Web. These nodes were added to Open Inventor in release 2.1, making Open Inventor a true superset of VRML. Release 2.1 of Open Inventor also introduced some new nodes and field types. VRML 2.0 continues to both borrow from and extend the Open Inventor model. VRML files are commonly called worlds and have the *.wrl* extension.

2.2.4.5 Stereolithography file format

The *stereolithography*[2] or STL file format is an ASCII or binary file widely used in *rapid prototyping* industry (see Palm [78]). It contains a list of the triangles that describe a solid model. This is the standard input for most rapid prototyping machines; most CAD packages allow to export to the STL file format. STL files have the *.stl* extension.

The structure of the STL file is extremely simple. It contains listings of individual triangles that define the faces of the solid model. Each individual triangle description defines a single normal vector directed away from the solid's surface followed by the coordinates for all three of the vertices (see STL specification [1]).

```
solid
 facet normal -1 0 0
  outer loop
   vertex 0 1 0
   vertex 0 0 0
   vertex 0 0 1
  endloop
 endfacet
 facet normal 0 0 -1
  outer loop
   vertex 0 0 0
   vertex 0 1 0
   vertex 1 0 0
  endloop
 endfacet
 facet normal 0.577 0.577 0.577
  outer loop
   vertex 0 0 1
   vertex 1 0 0
   vertex 0 1 0
  endloop
 endfacet
 facet normal 0 -1 0
  outer loop
   vertex 1 0 0
   vertex 0 0 1
   vertex 0 0 0
  endloop
 endfacet
endsolid
```

Listing 2.4: The model of tetrahedron stored in STL file format.

For us the most important difference between the STL format and the formats discussed previously is the fact that the STL file doesn't contain indexed face sets. What we have here is the so called *triangle soup*: there is no connectivity information, every vertex is repeated for each triangle adjacent to it.

Consequently, in order to handle STL files we have to perform the preprocessing phase, during which we read the input triangle soup and convert it to an *indexed face set* representation, where

- every vertex is stored only once, and
- the triangles are determined by the indices of according vertices.

[2] Stereolithography is the most widely used rapid prototyping technology.

We accomplish this by lexicographically sorting the vertices in a priority queue, based on their coordinates. Let v_{stored} be a stored vertex and v_{input}, the actual input vertex from the input triangle T_{input}. In the case the distance $d(v_{stored}, v_{input}) = ||v_{stored} - v_{input}|| < \varepsilon$ for some ε,[3] the vertices are considered to be the same, v_{input} is not added to the priority queue, and T_{input} is stored with the index of v_{stored}.

[3]Please note that in practice coordinates of the recurring vertices are exactly the same, that makes the choice of ε trivial.

Part II

Mesh repair

In this part, we present two algorithms for repair of polygonal meshes containing various artefacts due to incorrect modelling or acquisition. The first of the methods is dealing with the inconsistency of normal orientation (chapter 3), while the second, removes the inconsistency of vertex connectivity (chapter 4).

Chapter 3

Consistent orientation of normals

Boundary representations consist of a set of primitives, with or without topological information. Important examples of such primitives are polygons, patches generated from subdivision surfaces and NURBS patches. In many areas of computer graphics it is desirable or even necessary that the primitives of a model are consistently oriented, i.e. that the surface normals point in the "correct" direction.

One area, where consistent normal orientation is highly desirable, is *real-time rendering*. Figure 3.1 a shows an example of an inconsistently oriented polygonal model[1]. Inconsistent orientation results in incorrect lighting (the so-called "checkerboard effect"). This could be remedied by two-sided lighting or by doubling the number of light sources. However, both ways will decrease rendering performance. In addition, correct normals are still needed to perform back-face culling, a technique to further improve rendering performance.

Similarly, correct orientation of a model's primitives is important in *ray tracing* and *radiosity*, otherwise lighting artefacts will be caused.

More importantly, inconsistent orientation of normals can be fatal for many well-established mesh processing algorithms. For example, the mesh simplification algorithm proposed by Garland and Heckbert [30] uses vertex normals to determine the order in which contraction operations are to be performed. If applied to a model with inconsistent normals, this algorithm will produce severe artefacts.

Other areas are the computation of basic object properties, such as volume and mass, rapid prototyping [88], NC machining and the optimization of wireless communication systems [52].

[1] The bot model is courtesy of Michael Beals.

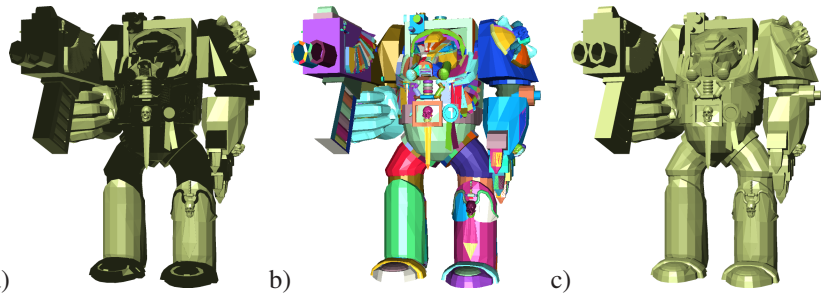


Figure 3.1: a) Original model of a bot consists of 2398 inconsistently oriented separate surface patches; due to the lighting, the "backwards"-oriented faces are rendered dark. b) Different manifold surface patches are rendered in different colours. c) The bot model after applying our algorithm (with exactly the same lighting).

Many models are not designed as solids, but just as a single sheet of patches (such as a windshield). In figure 3.1 b, separate surface patches, of which the bot model from figure 3.1 a consists, are shown in different colours.

Unfortunately, many modelling tools, in particular CAD tools, pay little attention to the consistency of normal orientation. There is no feature that automatically orients the normals, so designers have to manually orient each patch.[2] In addition, many models contain other geometric flaws, such as unintentionally intersecting primitives, cracks or gaps and T-junctions, which are discussed in more detail in section 4.

To solve the problem of inconsist normal orientation, we developed an algorithm that allows to consistently orient the normals of a boundary representation, even in the presence of gaps, T-junctions and intersections. The input of our algorithm consists of an arbitrary set of primitives, without any topology information. It can handle non-closed and even non-manifold objects (see definitions 2.5 and 2.6 from section 2.1.1.1).

Our method builds a connectivity graph of the patches of the model, which encodes the *proximity* of neighbouring patches. In addition, it augments this graph with two visibility coefficients for each patch. Based on this graph, a global consistent orientation of all patches is quickly found by a greedy optimization.

Figure 3.1 c shows the output of our algorithm for the bot model from figure 3.1 a, which now has consistent normals everywhere.

We describe our algorithm for polygonal objects only. Note, however, that it works just the same for objects consisting of NURBS or other primitives.

[2] At many German automotive companies, there are design guidelines that include rules how designers should orient surface patches, but it is often very difficult to enforce them.

Thus, tessellations from such models with consistent normals would also be consistent.

The results show that for almost all models our algorithm produces the desired normal orientation.

3.1 Description of the algorithm

The input to our algorithm is an arbitrary *polygon soup*, i.e. a set of unorganized polygons without any explicit topology information, with or without normals. Since the orientation of each polygon's normal can be encoded in its vertex order, the problem of orientation of normals is equivalent to the problem of ordering vertices in polygons consistently. In the following, we will treat the term *polygon normal* synonymous with the term *vertex order*.

Usually, the output from modelling tools consists of a number of *patches*. Here, a patch consists of a set of polygons that are connected to each other, i.e. for which topology information can be constructed trivially. However, between patches there are usually more or less wide gaps.

First, we build the neighbourhood information, divide the model into manifold surface patches and detect their boundaries. At this point we also orient the polygons consistently within each patch, which can be done trivially based on the topology information that is now available. After that, we determine those pairs of patches that are close to each other along some extent of their respective boundaries. For each such boundary pair we calculate its *coherence coefficient*.

Next, we determine the *visibility coefficients* for both sides of each patch. These coefficients describe how much of the surface of the patch is visible when viewed from all different viewing angles.

Finally, using both the boundary coherence and visibility coefficients we compute a global consistent orientation of the whole model. Patch boundaries that are close to each other make our algorithm consider normal orientation consistency more important than front-face visibility. On the other hand, patches that share only very loose boundaries are oriented such that front-face visibility is favored over normal consistency across the boundaries.

In the following, we will describe each step of the algorithm in detail.

3.1.1 Detection of patches

As already mentioned, the input of our algorithm is a set of unorganized polygons. First we read the input polygons and, using lexicographical sorting of the vertices, convert them to an indexed face set.

After that, we build the neighbourhood information for the mesh. In order to do so, we detect and collect all boundary and non-manifold edges. As

boundary edges we define all edges which are incident to only one polygon. As non-manifold edges we define all edges which are incident to more than two polygons. During the traversal of the mesh we divide it into a set of manifold surface *patches*, which either are not connected with each other or connected only at vertices or non-manifold edges. For each patch we consistently orient all polygons belonging to it, which is trivial, due to the manifold topology that we now have. Of course, after that, the orientation could differ between two neighbour patches, even if it was consistent in the original data.

The only problem that remains is how to orient the patches with respect to each other, which is not trivial, because their boundaries are only more or less close to each other along a part of that boundary.

3.1.2 Calculation of boundary coherence

At this stage we want to find close boundaries of different patches and determine the degree of their coherence. To accelerate finding pairs of close boundary edges we use a 3D grid, but many other spatial acceleration structures, such as k-d trees, can be used as well. Note that we should set a large search distance, as the quality of the results will suffer, if it is too small.

Assume that we have found the boundary edge e_j^n from patch P^n as a closest neighbour for a boundary edge e_i^m from patch P^m. Then, we proceed in the following way: first, we calculate the *local coherence* between these two edges, which we define as

$$c_{ij}^{mn} = -\operatorname{sgn}(s_{ij}^{mn}) \frac{\sqrt{|s_{ij}^{mn}|}}{1 + d_{ij}^{mn}}, \tag{3.1}$$

where $s_{ij}^{mn} = \vec{e}_i^m \cdot \vec{e}_j^n$ is the scalar product of $\vec{e}_i^m$ and $\vec{e}_j^n$, d_{ij}^{mn} is the shortest distance between e_i^m and e_j^n.[3] The absolute value of the local coherence is approximately proportional to the edges' lengths and inversely proportional to the distance between them. Its sign shows whether the polygons incident to these boundary edges have the same or different orientation.

All local coherences for edge pairs from the patches P^m and P^n are summed up into the *coherence coefficient*:

$$c^{mn} = \sum_{i,j} c_{ij}^{mn}. \tag{3.2}$$

Figure 3.2 shows an example of the coherence coefficients. The idea of the boundary coherence coefficient is that it can give a hint as to which patches should probably be oriented consistently. This is, of course, only an intrinsic constraint.

[3] Note that all lengths are normalized by dividing them by the length of the longest side of the model's bounding box.

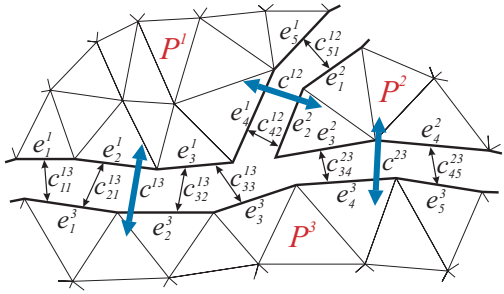


Figure 3.2: Local coherences c_{ij}^{mn} and coherence coefficients c^{mn} for patches P^1, P^2 and P^3.

3.1.3 Calculation of visibility

We still need an external indicator to help choose the correct overall orientation of all patches. As mentioned before, our goal is to find a global orientation of patches, such that as many polygons as possible can be seen with their front faces from most viewpoints. For this purpose, we want to determine the visibility of each side of each patch when seen from all possible viewpoints from outside the whole object. To do this, we can use various methods.

3.1.3.1 Ray shooting method

The first method we have tried is similar to a common raytracer. On the surface of each patch P^m we randomly and uniformly choose n^m points, where n^m is proportional to the area of P^m. Starting from each of these points we shoot a ray in a random direction. If the ray does not intersect any other polygons, we increment the counter for the polygon's side corresponding to the half-space in which the ray was shot. So, each patch has two counters n_f^m and n_b^m, which are the accumulation of all polygon counters. Finally, we define the *front-face* and *back-face visibility coefficients* for each patch as

$$v_f^m = \frac{n_f^m}{n^m}, \quad v_b^m = \frac{n_b^m}{n^m}. \tag{3.3}$$

The main drawback of this method is that one needs to shoot a very large amount of rays in order to obtain an acceptable reliability.

3.1.3.2 5D octree method

In the previous method we walk along a ray every time we shoot it into the scene. Of course, we accelerate this by any of the well-known data structures,

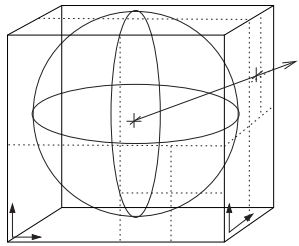


Figure 3.3: Ray space can be represented by a 5D cartesian rectangle with the help of the direction cube.

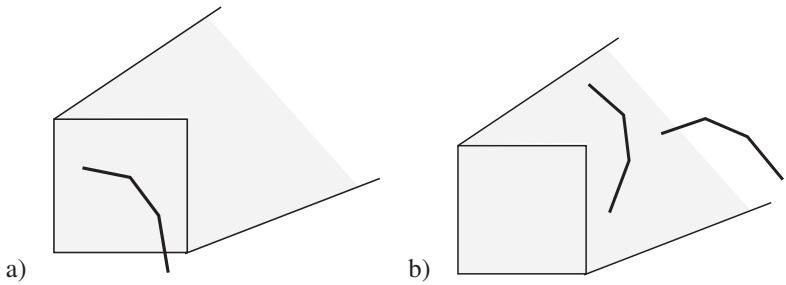


Figure 3.4: Two basic cases that can occur during computation of the visibility coefficients using a 5D octree over ray space.

such as k-d trees. However, rays are essentially static objects, just like the geometry of the model. So, based on the ideas of Arvo and Kirk [2], we develop the following algorithm.

We discretize directions by the so-called *direction cube* (see figure 3.3). Now we can build six octrees over the space of all rays, which are the 5-dimensional rectangles $R = U \times [-1,+1]^2 \times \{+x,-x,+y,-y,+z,-z\}$, where U is a suitable 3-dimensional bounding box around the complete model. Each cell, of the 5D octree corresponds to a beam in 3D geometric space, emanating from a 3D cell in geometric space [110].

Initially, we start with the root of the octree that comprises all possible rays. We associate all patches of our model with the root. Then, we recursively partition a node of the octree and distribute the set of patches among its children. A patch is associated with a child if it intersects the beam that child corresponds to. We stop the partitioning (i.e. conceptually we create a leaf), if either of the following conditions holds:

1. There is only one patch left in the 3D cell of the node. Now we must consider two sub-cases (see figure 3.4):

(a) There is no other patch associated with the node, i.e., all rays starting from the patch in the cube and in the direction of the beam would not hit any other patch (except for self-occlusions). Therefore, we add an amount to the visibility coefficient that corresponds to the spatial angle of the beam. Note that this angle depends only on the depth of the node, so the increase of the visibility coefficient can be precomputed.

(b) There are other patches associated with the node. This means that at least some rays shot from within the 3D cell would hit another patch. Since we are only interested in approximations of the visibility coefficient, we assume that all rays would hit, and therefore, we don't increase the coefficient.

2. The node's cell is too small (i.e., we have reached the maximum depth). Now we just consider the two sub-cases of the previous case for each of the patches that are (partially) inside the 3D cell of the node.

Note again that this computes, just like the previous of the following method, an approximation of the visibility coefficient. In our experience, though, such approximations are sufficient for all models we have tried so far.

Since we have now defined what a 5D cell of the octree represents, it is almost trivial to define how objects are assigned to sub-cells: we just compare the bounding volume of each object against the sub-cells 3D beam. Note that an object can be assigned to several sub-cells (just like in regular 3D octrees). The test whether or not an object intersects a beam could be simplified further by enclosing a beam within a cone and then checking the object's bounding sphere against that cone. This just increases the number of false positives a little bit.

Note that the octree does not have to be built at all. As soon as we arrive at a leaf, we possibly increase the visibility coefficient and then backtrack the recursion – we never actually construct any nodes. This greatly increases processing time. Furthermore, in contrast to Arvo and Krik [2], we need only very little memory, because we only keep some arrays of pointers to patches, one per recursion level. The number of levels is fairly limited (10–20).

3.1.3.3 GPU-based method

This method uses the GPU to calculate the visibility of single patches. The whole mesh is rendered from different points of view with colour coding, then the frame buffer is read and processed.

In order to get correct results we have to distribute the viewpoints uniformly around the model. We achieve this by placing them on the vertices of a tessellation of the bounding sphere of the model, produced by a successive subdivision of the icosahedron.

For each viewpoint we render the whole model onto a square viewport with a side length l_{vp}, using orthogonal projection, without shading and without anti-aliasing. Each side of each patch is drawn in an unique colour, which allows to unambiguously identify it when reading the pixels from the frame buffer. For each non-black pixel we increase the appropriate counter n_f^m or n_b^m (for front- or back-face, respectively) of the patch P^m by 1.

The side length l_{vp} should be chosen such that the smallest patch in the model will still occupy at least a few pixels from several viewpoints. Therefore, this method is only suitable for models where the ratio of the bounding box to the size of the smallest patch is not to large.

After n_t viewpoints, we define the *front-face* and *back-face visibility coefficients* for each patch P^m as

$$v_f^m = \frac{n_f^m}{n_t \cdot a^m}, \quad v_b^m = \frac{n_b^m}{n_t \cdot a^m}, \tag{3.4}$$

where a^m is the area of the patch P^m.

3.1.4 Consistent orientation of patches

After we have computed boundary coherence and visibility coefficients, we combine this information to find a consistent, global orientation of all surface patches.

For each patch P^m we already have its area a^m and two visibility coefficients v_f^m and v_b^m. We also have the set $\mathscr{C}$ of boundary coherence coefficients c^{mn}.

For each possible joint orientation of patches we define the overall front-face visibility V_f, back-face visibility V_b and coherence C of the super-patch as

$$V_f = \frac{\sum_m v_f^m \cdot a^m}{\sum_m a^m}, \quad V_b = \frac{\sum_m v_b^m \cdot a^m}{\sum_m a^m}, \quad C = \sum_{m,n} c^{mn}. \tag{3.5}$$

Our goal is to find the orientation of all patches that maximizes both overall front-face visibility V_f and overall coherence C for all super-patches.

We will now repeatedly replace the set of patches $\mathscr{P}$ by a new set $\mathscr{P}'$, where two former patches $P^k, P^l \in \mathscr{P}$ have been joined conceptually into a *super-patch* $P^j \in \mathscr{P}'$. During this join operation, the orientation of one or both patches can be flipped.

In the following we will denote by the word *patch* either an original patch or a number of patches joined into one super-patch.

When we flip the orientation of a patch P^k, we update the coherence and visibility coefficients related to this patch in the following way:

$$\begin{gathered} \hat{v}_f^k = v_b^k, \quad \hat{v}_b^k = v_f^k, \\ \forall c^{mk} : \hat{c}^{mk} = -c^{mk}, \quad \forall c^{kn} : \hat{c}^{kn} = -c^{kn} \end{gathered} \tag{3.6}$$

In order to achieve a fast algorithm, we use a greedy strategy: in a queue, we sort all pairs of patches for which the boundary coherence is defined, so that the absolute values of the coherence coefficients c^{mn} are sorted in descending order. Then, we connect pairs of patches into super-patches, which get their own visibility coefficients. We also compute new boundary coherence coefficients between the new super-patch and the other patches and insert them into the queue.

More specifically, with each step we take a pair of patches P^m and P^n with the largest absolute coherence coefficient c^{mn} out of the queue. Their visibility coefficients are v_f^m, v_b^m, v_f^n and v_b^n, respectively. Depending on a all these coefficients, we make a decision considering joint orientation of both patches.

3.1.4.1 Conforming coefficients

If

$$\begin{array}{llllllll} (c^{mn} > 0 & \wedge & v_f^m \geq v_b^m & \wedge & v_f^n \geq v_b^n) & \vee \\ (c^{mn} > 0 & \wedge & v_f^m \leq v_b^m & \wedge & v_f^n \leq v_b^n) & \vee \\ (c^{mn} < 0 & \wedge & v_f^m \geq v_b^m & \wedge & v_f^n \leq v_b^n) & \vee \\ (c^{mn} < 0 & \wedge & v_f^m \leq v_b^m & \wedge & v_f^n \geq v_b^n), & \end{array}$$

then the visibility coefficients agree with the coherence coefficients. Therefore, we connect both patches into one super-patch S and define its front-face and back-face visibility coefficients as

$$\begin{aligned} v_f &= \frac{v_{max}^m \cdot a^m + v_{max}^n \cdot a^n}{a^m + a^n}, \\ v_b &= \frac{v_{min}^m \cdot a^m + v_{min}^n \cdot a^n}{a^m + a^n}, \end{aligned} \tag{3.7}$$

where $v_{max}^m = \max(v_f^m, v_b^m)$, $v_{min}^m = \min(v_f^m, v_b^m)$, a^m and a^n are the areas of the patches P^m and P^n. We also change orientation of one or both patches, if necessary (if $v_f < v_b$). If c^{mn} was negative, it becomes positive after the change of orientation (according to equations 3.6.

This choice of orientations results in the maximization of the front-face visibility v_f of the super-patch S and, at the same time, its consistent orientation.

3.1.4.2 Conflicting coefficients

If

$$\begin{array}{llllllll} (c^{mn} > 0 & \wedge & v_f^m \geq v_b^m & \wedge & v_f^n \leq v_b^n) & \vee \\ (c^{mn} > 0 & \wedge & v_f^m \leq v_b^m & \wedge & v_f^n \geq v_b^n) & \vee \\ (c^{mn} < 0 & \wedge & v_f^m \geq v_b^m & \wedge & v_f^n \geq v_b^n) & \vee \\ (c^{mn} < 0 & \wedge & v_f^m \leq v_b^m & \wedge & v_f^n \leq v_b^n), & \end{array}$$

then the visibility coefficients come into conflict with the coherence coefficients: if we choose the patch orientations according to c^{mn}, the front-face visibility of the resulting super-patch will be not the maximum possible; on the other hand, the choice of orientations, which maximizes the front-face visibility, will result in inconsistent orientation on the boundary between the patches.

To find a tradeoff between the front-face visibility and boundary coherence, we compare them with some predefined values, which are parameters of the algorithm.

As already mentioned in the beginning of section 3.1, in case of close patch boundaries we consider normal orientation consistency more important than front-face visibility. Therefore, we first compare the boundary coherence with a threshold C_0. If $|c^{mn}/l^{mn}| > C_0$, where l^{mn} is the sum of lengths of all edges that contribute to c^{mn}, we assume the coherence between two patches to be strong and preserve their consistent orientation by connecting both patches into one super-patch. The visibility coefficients of the new super-patch are defined as

$$\begin{aligned} v_f = \max(v_1, v_2), \quad v_b = \min(v_1, v_2), \quad \text{where} \\ v_1 = \frac{v^m_{max} \cdot a^m + v^n_{min} \cdot a^n}{a^m + a^n}, \\ v_2 = \frac{v^m_{min} \cdot a^m + v^n_{max} \cdot a^n}{a^m + a^n}. \end{aligned} \tag{3.8}$$

If necessary, we change the orientation of one or both patches.

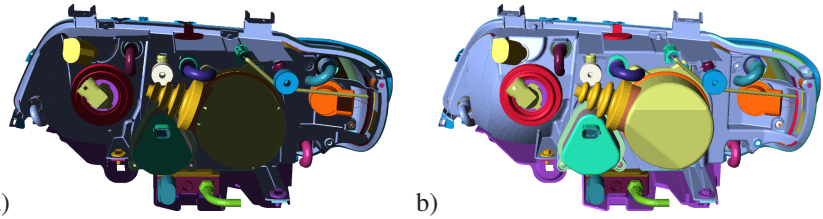
Otherwise, we compare the visibility coefficients off the two patches. If for one of the patches both visibility coefficients are very small or differ not much, and the visibility of the other patch dominates over it with respect to the patches' areas, we assume that its incorrect orientation will have only tiny impact on the overall front-face visibility. Therefore, if for the patch P^m

$$\left(v^m_b > \varepsilon_v \quad \wedge \quad \frac{v^m_f}{v^m_b} < k_v \quad \vee \quad v^m_b < \varepsilon_v \quad \wedge \quad v^m_f < \varepsilon_v\right) \quad \wedge \quad v_1 < v_2, \tag{3.9}$$

we connect both patches into one super-patch. Here, ε_v is a lower threshold of visibility, below which we assume it is not important; k_v is a minimum ratio of largest to smallest visibilities of a patch, below which we assume their incorrect orientation is not important; v_1 and v_2 are defined in equation 3.8. The visibility coefficients of the new super-patch are calculated according to equation 3.8. If necessary, we change the orientation of one or both patches.

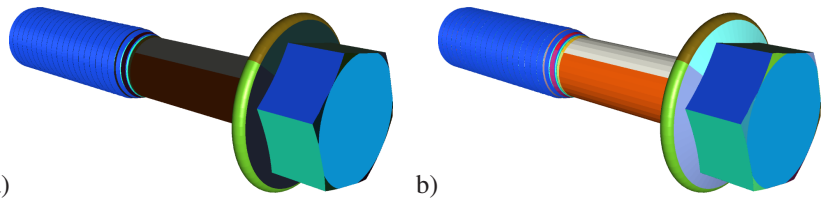
If condition 3.9 does not hold, we perform these comparisons again for the second patch P^n and, if true, still connect the two patches.

In all other cases we decide to favor front-face visibility over normal consistency across the boundaries and do not connect the patches. All their coefficients remain unchanged and we proceed to the next pair of patches from the priority queue.



a) b)

Figure 3.5: Model consisting of 323 patches, which are shown in different colours. Dark lighting shows back-facing polygons, which denote incorrect orientation: a) original model; b) after applying our algorithm, all patches are correctly oriented.



a) b)

Figure 3.6: Model with 78 patches: a) original model; b) after applying our algorithm.

After the whole queue is processed, we get the final global orientation of all surface patches.

3.2 Results

We have tested our method with a large number of models consisting of up to 3000 surface patches.

Our new algorithm produced the desired orientation for almost all models of our test suite. Figures 3.5 and 3.6 show two samples of our test suite. Our method can also handle the coffee mug model shown in figure 3.7, which Murali and Funkhouser reported to be difficult for *proximity-based* approaches [70].

Additionally, we tested the robustness of our algorithm. To this end, we changed the orientation of a random sample of the input polygons (see figure 3.7 a). This had no influence on the result, i.e., our algorithm does not depend on the initial orientation of the input and all normals are correct afterwards.

We have also investigated whether only one of the two criteria (boundary coherence or visibility) would be sufficient. Our tests have shown that for many models the independent maximization of front-face visibility for each

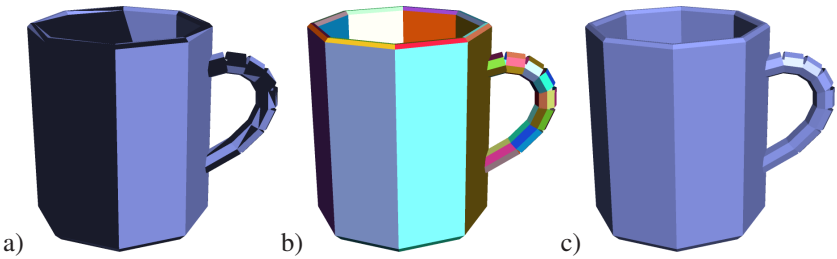


Figure 3.7: The coffee mug model used by Murali and Funkhouser [70]: a) polygons are oriented randomly (back-facing polygons are drawn in black); b) different manifold surface patches are rendered in different colours; c) after applying our algorithm, all polygons are oriented correctly.

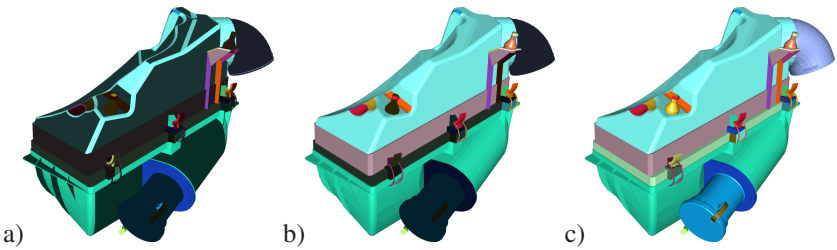


Figure 3.8: Model with 71 patches: a) original model; b) after applying our algorithm with only boundary coherence taken into account; c) taking both boundary coherence and visibility into account.

single patch can be sufficient. Using boundary coherence only was successful with some models. However, there are models where both criteria are needed. Figures 3.8 and 3.9 show examples where using only one of the two criteria fails to produce the desired result.

Table 3.1 shows the performance rates of our algorithm for three different models, each at three different levels of detail. To calculate the visibility coefficients we used the GPU-based method with 80 viewpoints and viewport of 400 x 400 pixels. The timings have been obtained on a Pentium 4 processor with 1.8 GHz and GeForce2 MX graphics card. Obviously, the overall times are dominated by the visibility computation. This time and also the time for building the topology information and detecting the patches are mostly linear in the number of polygons. The time for calculating the boundary coherence coefficients depends mostly on the number of boundary edges. Apparently, for the first model the simplification tool performed almost no reduction on boundaries.

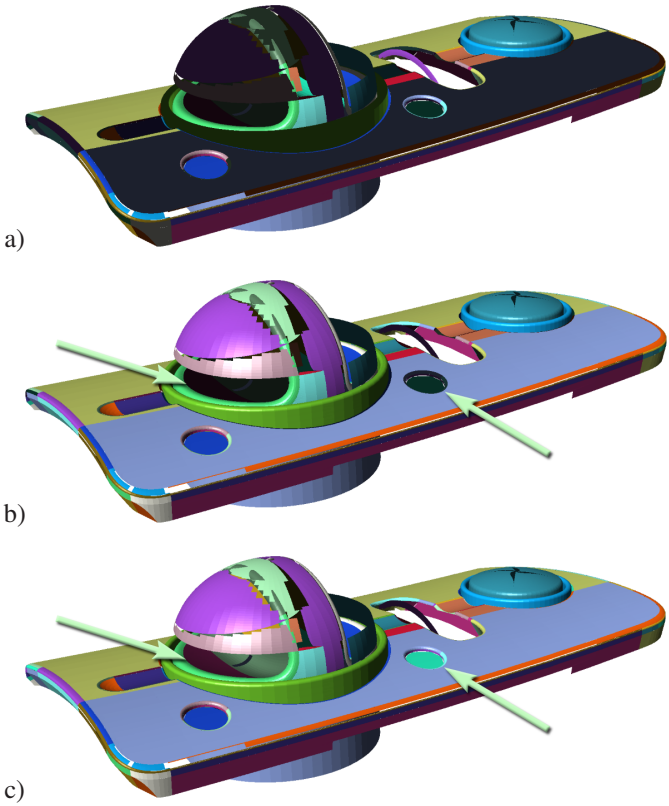


Figure 3.9: Model with 137 patches: a) original model; b) after applying our algorithm with only visibility taken into account; some incorrectly oriented patches are marked with arrows; c) taking both boundary coherence and visibility into account.

Figure 3.10: Model with 1250 patches: a) original model with patches shown in different colours and lighted from both sides; many patches overlap or are coplanar with each other; b) after applying our algorithm, several patches are still oriented incorrectly; c) the same model lighted from both sides.

	Number of polygons	Number of patches	Patch time (s)	Coherence time (s)	Visibility time (s)
1	180 252	78	5.6	0.7	17.7
	60 080	80	1.5	0.6	6.6
	18 019	83	0.4	0.5	2.5
2	194 668	1 508	5.9	3.4	19.0
	64 648	1 509	1.8	2.0	6.8
	19 142	1 511	0.4	1.1	2.6
3	300 836	3 310	10.5	9.2	29.1
	84 673	2 040	2.7	2.8	9.1
	14 327	2 067	0.4	0.9	2.5

Table 3.1: Performance rates of our algorithm for 3 models at different levels of detail: time for detecting the patches, time for calculating the boundary coherence coefficients and time for calculating the visibility coefficients.

It is difficult to compare the performance of our algorithm with that of Murali and Funkhouser without re-implementing their approach. Therefore, we tried to convert the timings reported by them on our platform. We used a scaling factor of 50, which means that their algorithm could handle a model of about 1200–1600 polygons in about 1.5–4.5 seconds. In contrast, our method can handle a model of 15 000–18 000 polygons in 3.4–4.1 seconds.

Since we do not only consider solids (i.e. objects that have, or should have, a well-defined interior and exterior), it should be mentioned here, that for some models a consistent orientation of all normals is not possible. The Möbius strip, shown in figure 3.11, is classical example of a *non-orientable* surface, as defined in definition 2.8 from section 2.1.1.1.

In addition, with (intentionally) non-manifold models, it can become very hard to define the best of all possible orientations, even for humans. But even in those cases, our method finds a good solution. Only for models with many overlapping and coplanar polygons it produces sub-optimal results (see figure 3.10).

3.3 Related work

So far, there were two approaches to solve the problem of inconsistent normal orientation: *proximity-based* or *boundary-based* and *solid-based*.

Proximity-based and boundary-based methods try to establish topological information based on the proximity of vertices or boundaries. In their surface reconstruction method, Hoppe et al. [47] determine a consistent orientation of tangent planes in all data points by solving a graph optimization problem. However, their method can be applied only to manifold models. Several works,

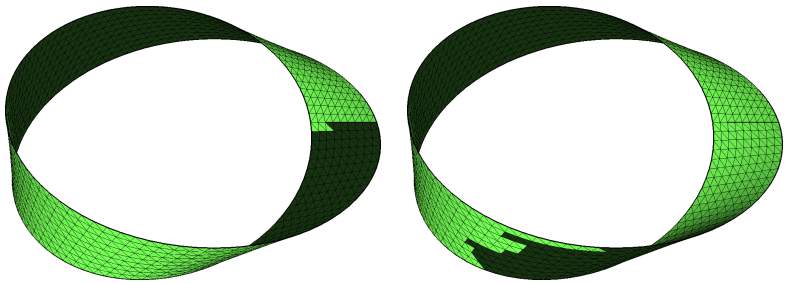


Figure 3.11: Some models cannot be oriented consistently. The Möbius strip is a simple example. Shown here are two different possible solutions.

e.g. by Bernardini et al. [10] or by Azernikov and Fischer [4, 6, 5], utilize a normal consistency criterion in order to correctly recover the topology of an object. Some other methods, e.g. by Kernighan and van Wyk [52] or by Laurini and Milleret-Raffort [57], are inherently two-dimensional.

Solid-based approaches, e.g. by Thibault and Naylor [93] or by Teller and Hanrahan [92], try to partition $\mathbb{R}^3$ into cells that are either inside or outside the model. Murali and Funkhouser [70] significantly extend this in order to deal with gaps, T-junctions and intersections. However, these methods can handle only geometry that is closed and manifold (or intended to possess these properties).

Guthe et al. [39] collect the patch connectivity information in a so-called *seam graph*, which is then used for view-dependent trimmed NURBS rendering.

In computational geometry, a lot of work has been devoted to robust computing (see works by Segal [86], Fortune and van Wyk [28] or Shewchuk [89]). However, these methods are not applicable here, because they try to avoid errors caused during the computation, while our algorithm tries to correct errors in the input data.

Since the publication of our method in the year 2004, several new approaches, which handle the problem of inconsistent normal orientation, were published.

Mesh repair methods based on a volumetric techniques, e.g. by Bischoff et al. [11], solve several mesh inconsistency problems simulteniously. The idea of these methods is to convert the mesh data into a volumetric representation and then generate a consistent mesh when converting back to a surface. In the volumetric setting, various filter operations can be applied to repair some of the topological artefacts and holes but this also removes sharp features from the input data. Hence, most volumetric approaches have been suggested in the context of topology-modifying mesh simplification.

The very recent visibility-based geometry-analysing algorithm by Zhou et al. [112] propagates visibility using ray casting and computes an inside and outside classification of oriented triangle faces using graph cuts. It is closely related to our method and improves several of its issues. Most notably, it is able to cope with coplanar polygons and intersections and incorporates a mechanism to correct sampling errors, which may appear due to the imperfection of the geometry, e.g. gaps between faces.

3.4 Summary

We have proposed a new approach, which we call *visibility-based*, to the problem of consistently and sensibly orienting all normals of arbitrary polygonal models. We combine this approach with a *proximity-based* approach, which yields a method that can correctly orient more models than previous methods.

We first divide the model into a set of manifold surface patches and establish a consistent orientation within each patch. Then, we orient the patches in such a way that the coherence between patches with close boundaries is maximized and, at the same time, as many polygons as possible are seen with their front-faces from most viewpoints.

Our method produces the desirable solution for almost all practical cases, except the models, which contain many overlapping and coplanar polygons.

The algorithm is controlled by only very few parameters, and their adjustment is not critical. While some of them balance the tradeoff between accuracy and speed, the others determine the choice between consistent orientation and visibility.

As we mentioned in the beginning of this chapter, our method is applicable not only to polygonal meshes, bot to objects consisting of other primitives as well, such as NURBS. This is one avenue for future work. Another area is the search of other metrics and other criteria for decision-making in conflict situations. Also, acceleration and increase of accuracy of our method could be considered as further development.

Chapter 4

Progressive gap closing

Since the generation of 3D models is application-driven, numerous models contain artefacts like T-vertices, degenerate triangles, gaps and holes. Essentially, a set of polygons not containing consistent connectivity information suffice for rendering purposes.

However, as a natural consequence of recent advances in computer graphics field, we no longer want to be able only to render images of objects, but also to process and analyze the already available models. New demands and applications have arisen, where "better behaved" polygonal models are desired, in a sense that they do not contain the above artefacts.

Signal processing techniques on meshes aim at analyzing the geometry and improving the visual quality of models. Compression and progressive transmission facilitate robust transfer of 3D meshes through the Internet and their efficient storage. Mesh simplification algorithms reduce the complexity of highly detailed models by optimally approximating the geometry within a prescribed tolerance.

The following example demonstrates the problem of applying a common mesh decimation algorithm to a model containing gaps between the patches. The model shown in figure 4.1 a consists of several separate patches with no visible gaps between the patch boundaries, which are rendered red in figure 4.1 b. However, after applying mesh simplification, the gaps can grow, as shown in figure 4.1 c.

Computing geodesic distances has gained a lot of attention in the last few years, since this numerical method was recently found very useful in a number of applications, e.g. in a work by Novotni and Klein [77]. The effect of lack of consistent connectivity of vertices is depicted in figure 4.2.

There are numerous variants of these geometric modelling techniques, and some of them are applicable to meshes containing the artefacts mentioned above. Generally however, in order to achieve optimal results, consistent vertex connectivity information is required.

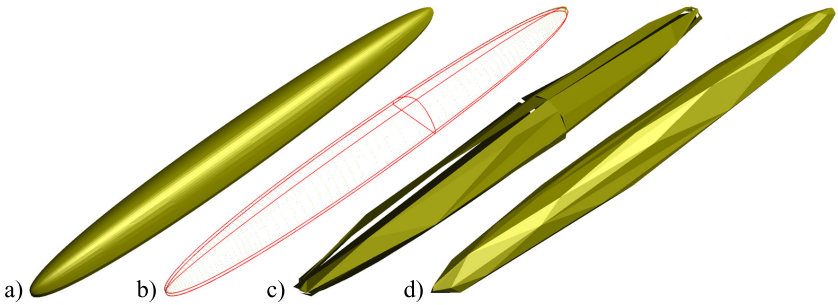


Figure 4.1: Impact of inconsistent vertex connectivity on a mesh simplification algorithm. The original model (a) contains gaps between separate patches, whose boundaries are rendered red in (b). After applying decimation, the gaps became visible (c). Simplification of the model repaired by our algorithm produces no such artefacts (d).

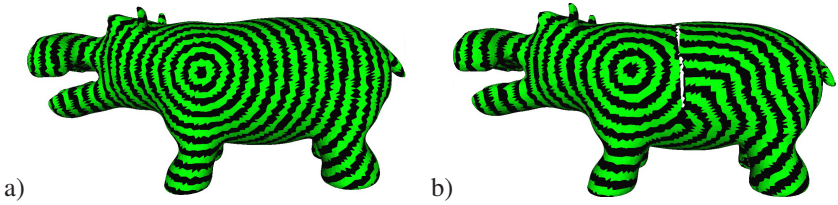


Figure 4.2: Impact of inconsistent vertex connectivity on an algorithm computing geodesic distances on triangular meshes. For a closed watertight hippo mesh (a), the algorithm produces appropriate wavefronts. In contrary, for the hippo that contains a long hole along its back (b), the wavefront breaks at the ends of the hole, thus producing erroneous results.

We developed a method that allows to eliminate these artefacts and repair the input mesh by applying topological modifications while retaining the overall geometry.

In the proposed method we suggest to approach the problem of eliminating the artefacts listed above as a simplification task. Without loss of generality, we formulate our technique for triangle meshes. Since we intend to proceed by altering the topology of the triangle mesh, we clearly need topology modifying operators.

Garland and Heckbert [29] and Popović and Hoppe [79] already generalized the common edge contraction operator, where two vertices lying on a common edge are contracted, to vertex contraction, where two vertices not necessarily connected by an edge are contracted. This way unconnected regions

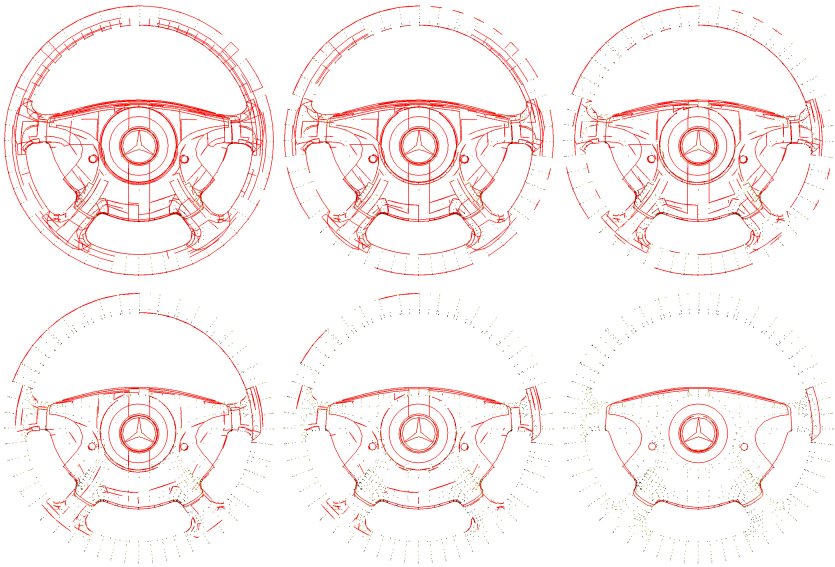


Figure 4.3: Representative stages of repairing the model of a steering wheel, demonstrating the progress of our algorithm. Only the boundary edges are rendered as wireframe, the vertices of the mesh are rendered to give a feeling where the surface is. About 2700 models were generated in sequence between the leftmost original mesh and the rightmost final one. After the generation of this sequence, the user is free to navigate back and forth between the models and select the desired one, she/he can also choose to proceed further with the gap closing.

of the mesh can be joined. We further generalize this operation by introducing a *vertex-edge* contraction, where a vertex is unified with its projection on an edge.

The intuition behind our algorithm is that the gap closing should proceed by attracting the boundaries of the mesh towards each other, which is achieved by utilizing the vertex-edge contraction operator. In other words, we apply the methodology of mesh decimation to the decimation of boundaries targeted at gap closing.

Similarly to the common simplification methods, the procedure is error-controlled as well, furthermore, it is performed in a progressive manner according to a monotonically increasing error. Our method essentially generates a sequence of meshes where the gaps and holes are progressively removed, as shown in figure 4.3.

As explained in section 4.1, the transition between the neighbouring models in the sequence is accomplished by applying a single contraction operator or its inverse. We enable the user to navigate between these meshes, which facilitates the visual inspection of the results and interactive control of the process by determining a desired error tolerance.

As pointed out by Weihe and Willhalm [97], stitching of mesh boundaries is a highly non-trivial process, since some of the gaps may be intentionally modelled, while others might be results of erroneous modelling or tessellation procedure. We take this issue into account by allowing the user to manually select/deselect areas to be considered during the stitching process.

Again, the input to our algorithm is a general *polygon soup* – a set of unorganized polygons without any explicit topology information, à la STL file format (see section 2.2.4.5). We assume the mesh to be composed of triangular faces. Note that this does not imply any loss of generality, since the polygonal faces may easily be converted into triangular ones. The mesh to be processed by our method will possibly include the following artefacts:

- *degenerate faces* without finite area,
- unwanted *gaps* and *cracks* between regions of the mesh resulting from erroneous scan data reconstruction or modelling and/or tessellation of analytical surfaces,
- *holes* in the model due to missing polygons,
- *T-vertices* lying on interior of an edge of a face.

The vertex-edge contraction operator to be defined in section 4.1 will possibly introduce non-manifold edges and/or vertices (see definition 2.6 from section 2.1.1.1), and therefore, the output of our system will also be a non-manifold mesh in the general case. However, if a manifold surface is desired, the methods presented by Guéziec et al. [35] may be applied to our results; the mesh will be cut exactly along non-manifold features, thus preserving the consistent connectivity of the manifold components.

4.1 Vertex-edge contraction

Let us describe our notation and terminology: $\mathscr{V}$ is a set of N abstract vertices, $|\mathscr{V}| = N$. The abstract polygonal surface $\mathscr{S}(\mathscr{V})$ contains the topological information of the model, it is composed of subsets of $\mathscr{V}$. Since we work with triangle meshes, the subsets are vertices, edges and triangles. In order to embed the mesh into the three dimensional space $\mathbb{R}^3$, we assign a geometric position in space to each abstract vertex. Let $\mathscr{P} = \{p_i \in \mathbb{R}^3 | 1 \leq i \leq N\}$. We now define the triangular mesh $\mathscr{M}$ as the pair $(\mathscr{S}, \mathscr{P})$.

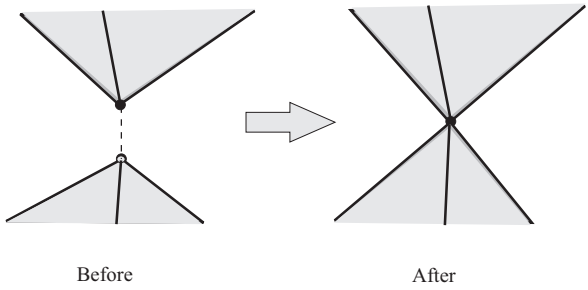


Figure 4.4: Vertex contraction operation.

The decimation methods utilizing the *edge contraction* operator proceed by iteratively contracting edges. As already pointed out above, this operator does not provide enough topological flexibility during the decimation process. It is possible to close holes in the mesh by iterative application of the operator, however, the disconnected regions of the mesh will remain in separate components. The *vertex contraction* operator (see figure 4.4) is a natural generalization of the edge contraction, and it seems to be the appropriate compromise between topological flexibility and control over the topological changes it induces. This operator contracts vertices not necessarily lying on a common edge, and therefore, it allows to stitch together the boundaries of the mesh. Note that we only want to process the boundaries of the mesh, in which case even the vertex contraction may be not flexible enough. In order not to introduce distortions in case of narrow gaps, it is sometimes more favorable to project a vertex directly onto an edge. We call this operator a *vertex-edge contraction*, which is made up of the sequence of following operations (see figure 4.5):

1. The vertex v is projected orthogonally onto the edge e.

2. A vertex v' is inserted into the edge at the geometric position of the projection.

3. The triangle t_1 is split into two triangles $t_1' = (v_0, v', v_2)$ and $t_2 = (v_1, v_2, v')$.

4. A vertex contraction to v and v' is performed. The new position of the vertex will be a convex combination of v and v', we move the new vertex v_{new} into position $\lambda v + (1-\lambda)v'$.

Please note that general version of the *vertex-edge contraction* operation, which dealing not only with mesh boundaries but with arbitrary vertex-edge pairs, will be discussed later in section 5.1.1, as we will introduce the generalized pair contraction operator.

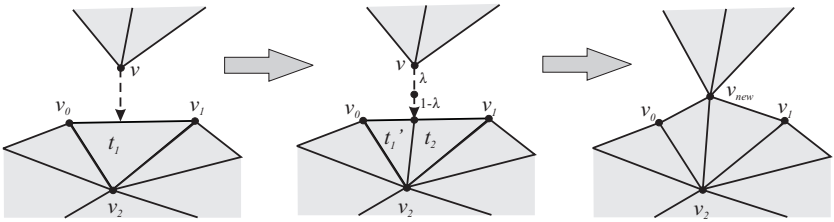


Figure 4.5: Vertex-edge contraction operation.

At this point, it is necessary to mention that during the process we maintain for each boundary edge a list of vertices being projected onto it. In case the geometric position of the edge changes, the edge is destroyed or no longer belongs to the boundary due to modifications in its vicinity, we recompute the correspondences for all vertices in the list. Furthermore, note that if the vertex v is projected onto a vertex incident to the edge e, only a simple vertex contraction is performed. Thus, the vertex-edge contraction is a further generalization of the vertex contraction. The projection of vertices is conducted according to an error measure, see the next section for details on that.

The goal of our method is to construct a series of meshes $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_M$ by incrementally applying the vertex-edge contraction operator, where $\mathcal{M}_0$ is the input mesh. The user should be able to choose the desired model $\mathcal{M}_i$, and therefore, we allow him/her to navigate between the meshes. Thus, we essentially generate a sequence of meshes in a sense of multiresolution representations, see e.g. Hoppe [44]. The forward navigation is clearly accomplished by applying the contraction according to some well defined order. However, in order to undo the operations during the backward navigation, we have to define an *inverse* vertex-edge contraction operator and store some data during the generation of the sequence. The data to be stored and the inverse operators for vertex contraction are described by Garland and Heckbert [29] and by Popović and Hoppe [79]. We focus only on formulating an inverse operator of the vertex-edge contraction (see figure 4.5):

1. The vertex v_{new} is projected orthogonally onto the edge e determined by vertices v_0 and v_1, this way we reconstruct the position of the vertex v'.

2. The vertex v is computed as follows: $v = \frac{v_{new} - \lambda v'}{1-\lambda}$.

3. The triangle t_2, which can unambiguously be determined, since we define it to be incident to v_1 and non-incident to v_0, is deleted.

4. The triangle $t_1 = (v_0, v_1, v_2)$ is restored. Note that v_2 can also always unambiguously be determined, since e is a boundary edge.

Note that we do not have to delete any vertex, since we reuse v to store v_{new}. Given a mesh $\mathscr{M}_i$, $0 < i \leq M$, in order to fully restore the mesh $\mathscr{M}_{i-1}$, we only need to store the indices of vertices v_0 and v_1 as *split information*. The projection direction is the vector pointing from v_{new} towards v'; in order not to be forced to recompute the projection, we additionally store the λ.

We store the split information for each projected boundary vertex; to retain the ordering, we include the indices of vertices of this kind in an array as the decimation proceeds. Moreover, note that in order to navigate between the already generated meshes, it is not necessary to maintain for every boundary edge lists of vertices corresponding to it. The ordering of features to be contracted, which we store in an array during the procedure, fully determines the process for these cases. We only have to store these information for the mesh $\mathscr{M}_M$, where M is the largest index in the mesh sequence, since the correspondences are needed only if the user chooses to continue to generate further meshes.

4.2 Description of the algorithm

The algorithm for gap closing essentially consists of a preprocessing phase and the boundary decimation process itself. The method proceeds according to an increasing error computed as distance between feature pairs that are possible candidates for a contraction operation. This in turn implies that our approach has the nice *progressive* property, which means that always the contraction corresponding to the smallest error is performed. The progressivity is not only a numerically pleasant feature, it is also greatly appealing on the user level, since the user can follow the process in an intuitive manner.

4.2.1 Preprocessing phase

During the preprocessing phase we first read the mesh and convert it to an *indexed face set* representation, if necessary. After that, we find all boundary edges and vertices of the input mesh and for each boundary vertex identify a *corresponding pair*, i.e. another boundary vertex or boundary edge, which will be used in the boundary decimation process. Finally, we order all pairs in a queue.

4.2.1.1 Reading the mesh

We first read the input triangle soup and convert it to an *indexed face set* representation where every vertex is stored only once, and the triangles are determined by the indices of according vertices. We accomplish this by lexicographically sorting the vertices in a priority queue. Let v_{stored} be a stored

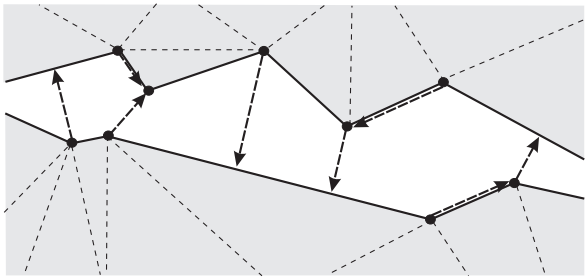


Figure 4.6: Result of the preprocessing phase; the dashed arrows indicate the correspondences. Note that some arrows point along an edge, which possibly implies an edge contraction.

vertex and v_{input} the actual input vertex, in case the distance $d(v_{stored}, v_{input}) = ||v_{stored} - v_{input}|| < \varepsilon$ for some ε, the vertices are considered to be the same and v_{input} is not added to the priority queue.

4.2.1.2 Identification of boundaries

Find all the edges e and vertices v that are elements of the boundary B. The boundary edges are those having only one incident triangle, the boundary vertices are simply the vertices incident to boundary edges.

4.2.1.3 Identification of corresponding pairs

In order to accomplish a pairing between vertices and edges to be contracted, for all boundary vertices we find the nearest boundary edge that is non-incident to the vertex. If an orthogonal projection of the examined vertex onto the corresponding nearest edge is possible, we store the edge as the paired feature, otherwise we store the nearest vertex of the edge. Additionally, for each boundary edge we store all corresponding vertices. Figure 4.6 shows fragments of the two close mesh boundaries with the identified correspondences marked by arrows.

4.2.1.4 Ordering of the pairs

For each feature pair we compute the distance between the features as an error measure. We subsequently include all the pairs into a priority queue sorted by this error.

4.2.2 Decimation step

For a boundary decimation step we first pop the vertex with minimal error from the queue, then we perform a vertex or vertex-edge contraction depending on the type of the corresponding nearest feature. Finally we maintain the correspondences for each vertex corresponding to a modified edge; we accordingly maintain the priority queue as well. The pseudo code for the decimation step can be found in algorithm 1.

Algorithm 1 Decimation step

```
1:  Vertex v = pqueue.pop_min();
2:  Feature f = v.nearest_feature();
3:  if is_vertex(f) then
4:      vertex_contraction(v,f);
5:  else
6:      vertex_edge_contraction(v,f);
7:  end if
8:  EdgeSet E = {modified_edges()};
9:  for all Edge e ∈ E do
10:     for all Vertex v ∈ {e.corresponding_features()} do
11:         v.maintain_correspondences();
12:         pqueue.reinsert(v);
13:     end for
14: end for
```

Note that if a vertex v is projected onto an edge in the vicinity of one of its vertices v_i, it is reasonable to *snap* the projected vertex v' to the vertex v_i, this way creating triangles with very small edges will be avoided. In our implementation if $d(v', v_i) < \varepsilon$, where ε is the global error threshold, we snap the vertices v_i and v'. Thus, in this case a vertex contraction will be applied to v and v_i.

4.3 Results

The first example shown in this section demonstrates a steering wheel model[1]. Shown in figures 4.7a and 4.7c, this model gives an insight into artefacts resulting from tessellating a complex model of a tessellated trimmed NURBS surface. Every kind of artefacts listed in the introduction can be found in this model. As it can be seen in the close-up in figure 4.7 a, there are holes even inside relatively flat triangular regions and narrow gaps between the original NURBS patches as well.

[1] The steering wheel model was kindly provided by DaimlerChrysler AG

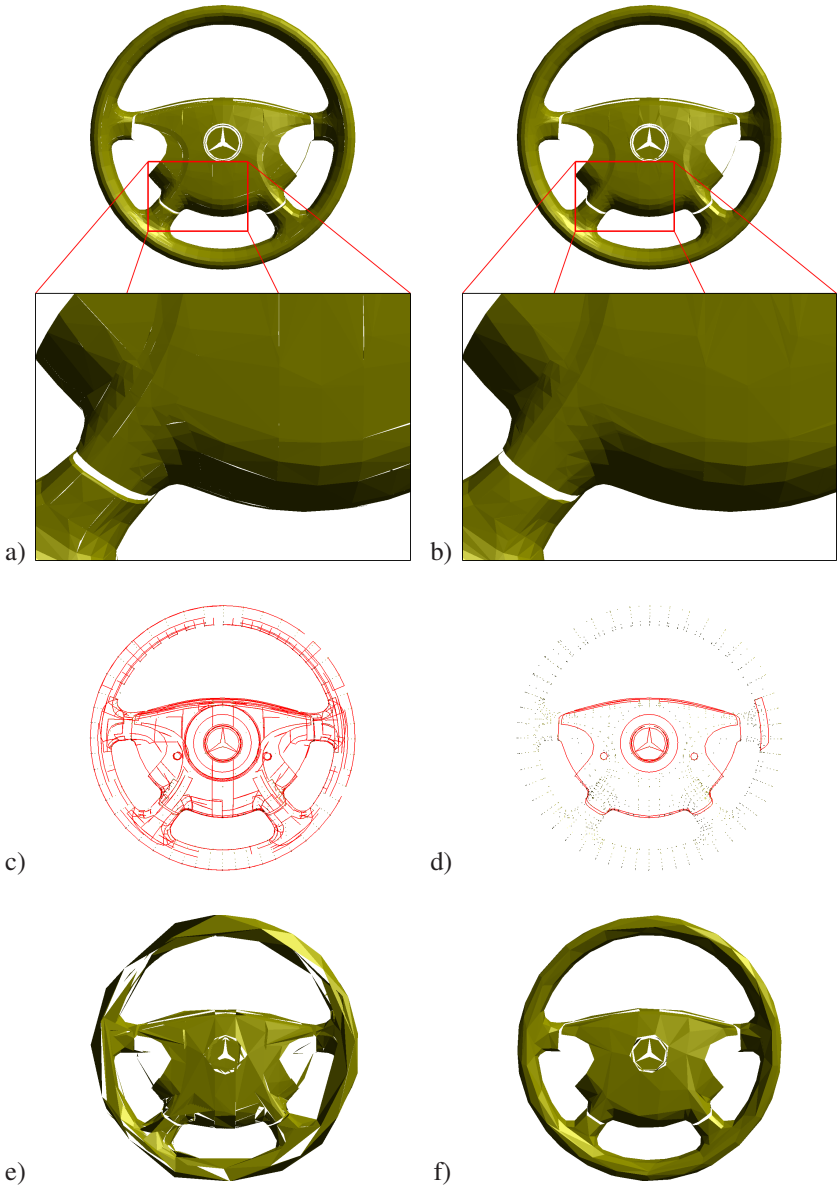


Figure 4.7: Results for the steering wheel model. The narrow gaps in the original model (a) have been removed by our procedure (b), which is even more obvious by looking at the boundaries (c and d). The images (e) and (f) demonstrate the impact on the decimation of models.

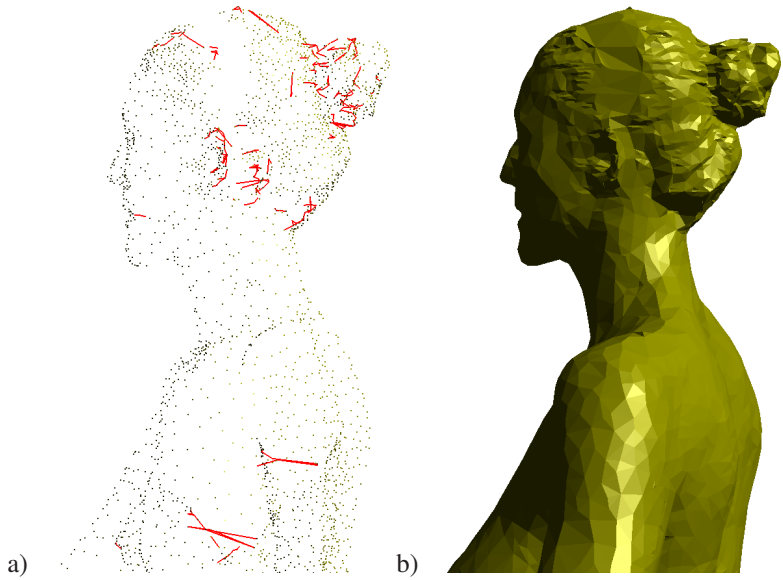


Figure 4.8: Results for a 3D laser scan. The boundaries identified in the original model (a) are depicted in red. Our procedure removed all the holes (b).

We managed to close the undesired gaps and holes, and the repaired model is shown in figures 4.7b and 4.7d. Note however, that the gaps intentionally modelled by the CAD tool operator are preserved.

The impact of the mesh repairing on performance of mesh processing methods is demonstrated by an example of triangle mesh simplification. The simplified original model (shown in figure 4.7 e) is hardly recognizable even after a relatively small amount of decimation steps. In contrary, the repaired model (shown in figure 4.7 f) retains the overall shape after a considerable reduction of number of triangles (the original model contains 6540 triangles, and it has been simplified to 1229 triangles).

An example of a different nature coming from a 3D laser scanner is the model of a woman shown in figure 4.8. The model consists basically of one connected component. However, it contains holes with jagged boundary and T-vertices. In figure 4.8 a all boundary edges are rendered in red, and the remaining model is rendered as points. As shown in figure 4.8 b, our method succeeds to remove all artefacts from this mesh as well.

4.4 Related work

Considerable amount of research and development has been conducted in the area of polygonal mesh and CAD data repair. Due to differences in inherent structures of meshes generated by various modelling tools and 3D acquisition techniques, the approaches handling the errors and degeneracies vary depending on the source of the data. Botsch et al. [12] provide an overview on the typical types of artefacts that occur in geometric models and introduce the most common algorithms that address these artefacts. Another survey of mesh errors and techniques for their detection and correction is done by Veleba and Felkel [96].

Turk and Levoy generate polygonal models from registered range data, they remove overlaps by clipping them, utilising a technique called mesh zippering [95]. The meshes coming from 3D scanners and volumetric data often contain artefacts of the reconstruction process: small handles and tunnels. Guskov and Wood conceptualized these as *topological noise*, identified and eliminated them by cutting and sealing the mesh, thus reducing the genus and topological complexity of the model [37].

Due to the industrial relevance of the problem, a lot of work has been devoted to repairing polygonal models generated by modelling tools, mainly CAD systems. Barequet and Kumar determine corresponding edges within an error tolerance and stitch them together in one pass [7]. Butlin and Stops present a method for repairing CAD data for analysis and exchange purposes [13]. Guéziec et al. generate manifold surfaces from non-manifold sets of polygons by identifying the topological singularities and decomposing the model into manifold components by cutting along these singularities [35]. They also describe a stitching operation allowing to join the boundaries of the components while guaranteeing the manifoldness.

Murali and Funkhouser first classify the regions of space as either solid or not and then generate a consistent set of polygons describing the boundary of the solids [70]. Nooruddin and Turk repair the polygonal models by converting them into volumetric representation, subsequently eliminate the topological noise by morphological open and close operators and finally reconstruct the polygonal mesh of the so-defined implicit function [75, 76].

Our gap-closing algorithm differs from the available techniques as it employs a well-established operation borrowed from *mesh simplification* field and as it is progressive. Since mesh simplification is one of fundamental operations on polygonal meshes, there is an extensive amount of literature on this topic, an overview of which will be given in section 5.7. Here, we will only mention the *vertex contraction* operation, which is directly related to our method. Introduced simultaneously by Popović and Hoppe [79] and Garland and Heckbert [29], it allows to contract any two vertices independently of whether they

are topologically adjacent or just geometrically close. Compared to edge collapse operation, proposed by Hoppe et al. [48], the vertex contraction offers more control over the topological modifications but is not general enough to connect close or even intersecting surfaces with small error early in the simplification.

Since the publication of our gap closing algorithm in the year 2002, many new methods related to mesh repair were published. Several recent approaches, e.g. by Ju [51] or by Bischoff et al. [11], are based on volumetric techniques (see section 3.3). The latter method overcomes the main disadvantage of the voxelization, i.e. giving away the original model, by keeping the vertex coordinates the same in the corrected model as in the input model.

Srinivasan et al. [90] proposed a purely topological algorithm to construct manifold meshes from arbitrary collections of polygons, which automatically and correctly creates the missing faces of manifolds with boundaries and eliminates several other mesh errors.

4.5 Summary

We presented a method that removes the inconsistency of vertex connectivity and thus produces high-fidelity models with properties, which are important prerequisites for most of the geometry processing and numerical simulation methods.

Essentially, our technique accomplishes the removal of undesired artefacts by simplifying the boundary of the mesh. For this purpose we generalized the vertex contraction operator by introducing the vertex-edge contraction operator, which operates on mesh boundaries.

The necessary topological modifications were already possible by applying the vertex contraction, however, our vertex-edge contraction provides additional flexibility. Our system is capable of creating a sequence of meshes generated during the boundary decimation process, which allows the user to choose the desired model. With additional interactive functionality, the user is enabled to select the regions of the mesh she/he wants to repair.

As for further development, we see plenty of room to develop interactive tools facilitating the effective work with the system. A 3D brush for instance could be used to navigate along a narrow gap and selecting it for subsequent *zippering*.

Part III

Mesh simplification

In this part, we present three algorithms dedicated to three different aspects of mesh simplification. The first of the methods tries to perform mesh simplification with the highest quality (chapter 5), the second, applies the developed technique to out-of-core simplification (chapter 6), and, finally, the third, prevents self-intersections during mesh simplification (chapter 7).

Chapter 5

High-quality simplification

Since the generation of 3D models is application-driven and mostly automatic, numerous models do not have consistent connectivity information. Typical artifacts in these models are T-vertices, degenerate triangles, self-intersections, gaps, small holes or very close but topologically unconnected surface parts. During simplification the artifacts can lead to unnecessarily large errors or even to further self-intersections in the simplified model. The vertex contraction operation introduced independently by Popović and Hoppe [79] and Garland and Heckbert [29] allows to contract any two vertices independently of whether they are topologically adjacent or just geometrically close. The vertex contraction facilitates topological modifications but is not general enough to connect close or even intersecting surfaces with small error early in the simplification.

In the previous part of this thesis, in chapter 4, we generalized the vertex contraction operation by contracting a vertex with an edge. This improves the sewing potential of the vertex-contraction-based simplification algorithm. Here we completely generalize the pair contraction approach and allow contraction of a vertex with another vertex, with an edge or a triangle and also contration of two edges. The new operations allow to connect close and intersecting surface parts that are not topologically incident on the early stages of simplification.

An important issue in the repair process is the ability of the user to specify the exact size of features like holes, gaps and cracks that should be removed from the model and, of course, in an intuitive solution small features should disappear before larger features. Using our new generalized pair contraction algorithm this can easily be achieved. During the simplification process, features are removed in the order of increasing *Hausdorff distance* between the original and simplified mesh, and therefore, according to the size of the features themselves. In order to repair a mesh, the user can specify a maximum size of features that should be removed and then simplification operations are subsequently performed until the specified threshold is reached. An example

is given in figure 5.6, where gaps and cracks in the triangulation of a steering wheel are successively removed and holes are closed in the expected manner.

This chapter is structured as follows. First, we introduce the generalized pair contraction operations. Then, in section 5.2, we explain the use of the quadric error metric with generalized pair contractions and describe improvements, which are necessary for the preservation of sharp features. Section 5.3 details our use of a spatial grid, which is used to find all potential contraction pairs. In section 5.4, we decribe the simplification algorithm, and then, in section 5.6, present results that demonstrate the advantages of the generalized pair contractions. Finally, we discuss related work and conclude.

5.1 Generalized pair contractions

As mentioned before (see section 4.4), the vertex contraction operation not always sews together geometrically close but not incident surface parts. In some cases a vertex on one part of a mesh is close to another part, but too far from any vertex on that part. In this case any vertex contraction between the two parts would introduce distortions and often also a large geometrical error. Sometimes it is more favorable to contract a vertex directly with an edge or triangle, what allows to connect the closest parts of a mesh and to close the most narrow gaps first.

But in some cases even this will be not sufficient. Two unconnected regions of a mesh, which are very close to each other, not necessarily have any vertices close enough to the other part in order to connect the parts without producing distortions. To enable sewing in such situations we also introduce a contraction operation that connects two close edges.

In summary, we extend the vertex contraction operator to the *generalized pair contraction* operator by introducing the new types of contractions: *vertex-edge*, *vertex-triangle* and, finally, *edge-edge*.

In the vertex-edge and vertex-triangle contraction operations, an *intermediate vertex* v' is created on the edge or triangle of the contraction pair just in order to contract it with the vertex v of the contraction pair. The latter is called *contraction vertex*. See figures 5.1 and 5.2 for an illustration. In case of the edge-edge contraction operation we create two *intermediate vertices* as shown in figure 5.3. In following sections we will describe the new contraction operations in more detail.

5.1.1 Vertex-edge contraction

The exterior case of this operation, when both the vertex and the edge lie on boundaries, was described in section 4.1. Now we allow pairs of an arbitrary vertex and an arbitrary edge. We proceed as follows (see figure 5.1):

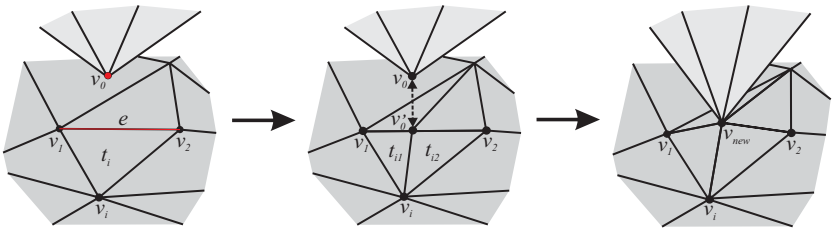


Figure 5.1: The vertex-edge contraction operation performs no reduction, but increases the connectedness of the model.

- Project the contraction vertex v_0 onto the edge $e = (v_1, v_2)$.
- Insert the intermediate vertex v'_0 on the edge at the geometric position of the projection and interpolate its quadric (see section 5.2).
- Split each triangle t_i, incident to the edge e, into two triangles $t_{i1} = (v_1, v'_0, v_i)$ and $t_{i2} = (v_2, v'_0, v_i)$.
- Perform a vertex contraction of v_0 and v'_0.

This operator perfectly allows to sew borders and close parts of the mesh. It doesn't decrease the number of vertices but increases the connectedness of the model. In the case, when both the vertex and the edge are incident to the same triangle, the vertex-edge contraction is topologically equivalent to an edge flip.

In the manifold case (see definition 2.6 from section 2.1.1.1) a second flip of the resulting edge would re-produce the original configuration. This can easily lead to an infinite number of successive edge flip operations. To avoid this problem we allow the edge flip, only if an additional criterion is fulfilled: the minimum angle among all affected triangles has to increase after the operation by a non-zero constant. As the minimum angle cannot increase by a constant infinitely, we avoid infinite sequences of edge flips. At the same time, triangles with acute angles can be removed, what improves the overall triangle shape.

5.1.2 Vertex-triangle contraction

Here we generalize the vertex contraction further by connecting an arbitrary vertex with an arbitrary non-incident face. While the vertex-edge contraction operation is chosen by the algorithm mostly on boundaries, this one connects geometrically close surface parts. The vertex-triangle contraction can be split in four steps (see figure 5.2):

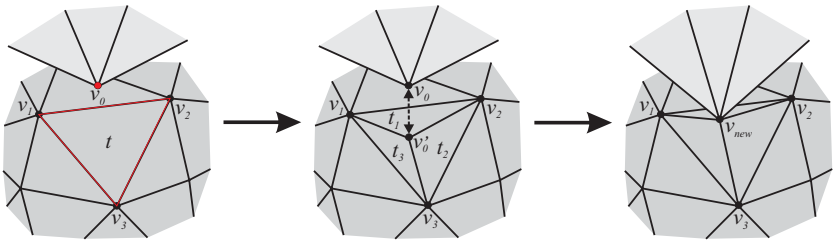


Figure 5.2: The vertex-triangle contraction operation performs no reduction, but increases the connectedness of the model.

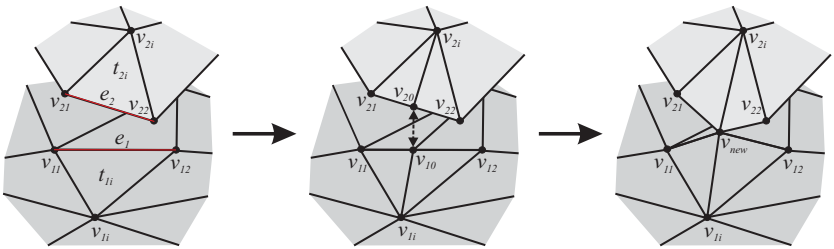


Figure 5.3: The edge-edge contraction operation increases the connectedness of the model by cost of the insertion of one vertex.

- Project the contraction vertex v_0 onto the triangle $t = (v_1, v_2, v_3)$.
- Insert the intermediate vertex v_0' on t at the projection point and interpolate its quadric (see section 5.2).
- Split the triangle t into three triangles $t_1 = (v_1, v_2, v_0')$, $t_2 = (v_2, v_3, v_0')$ and $t_3 = (v_3, v_1, v_0')$.
- Contract the vertices v_0 and v_0'.

The vertex-triangle contraction operator neither performs a reduction, but does increase the connectedness of the mesh.

5.1.3 Edge-edge contraction

The third generalized pair contraction operation is useful only in such cases, when two surface parts are close to each other or intersecting, but the distance between any vertex from one part to another part is significantly larger than the distance between two edges.

We proceed as in the vertex-edge contraction, but insert intermediate vertices on both edges (see figure 5.3):

- Find the shortest distance between edges $e_1 = (v_{11}, v_{12})$ and $e_2 = (v_{21}, v_{22})$.
- Insert the intermediate vertices v_{10} and v_{20} at the projection points and interpolate their quadrics (see section 5.2).
- Split each triangle t_{1i} incident to the edge e_1 into two triangles.
- Split each triangle t_{2i} incident to the edge e_2 into two triangles.
- Contract the vertices v_{10} and v_{20}.

We allow edge-edge contraction, only if the projection points lie inside the edges.

5.2 Order of operations

To order the possible contraction operations and to find the optimal position of a new vertex after the operation, our simplification algorithm uses the technique of *quadric error metrics* presented by Garland and Heckbert [29].

For each face f of the original mesh a *fundamental error quadric* $Q_f(p)$ is defined as the symmetric homogeneous 4×4 matrix, which measures the squared distance d^2 of a point $p \in \mathbb{R}^3$ to the plane of f as $d^2 = (p,1)Q_f(p,1)^t$. Each vertex v in the original mesh is assigned an initial quadric constructed as the matrix sum of the fundamental quadrics of its incident faces, divided by the order of v and optionally weighted by their areas.

Then, for each possible contraction of vertices v_1 and v_2, the vertex quadrics are added yielding the quadric $Q = Q_1 + Q_2$, which computes the sum over the two surface patches incident to v_1 and v_2 of the squared distances, divided by the order of the appropriate vertex and optionally area-weighted. The location of the new vertex v_{new} is set in a way to minimize the quadric error $e_q = v^T Q v$ caused by the performed contraction operation.

Finally, all possible contraction operations are ordered in a *priority queue*, with the quadric error e_q used as a key.

5.2.1 Non-accumulating error quadrics

In our approach, unlike the original algorithm by Garland and Heckbert, quadric errors are not accumulated, i.e. after performing the contraction operation the quadric $Q = Q_1 + Q_2$ is not associated with the newly created vertex v_{new}. Instead, the error quadrics are always calculated on base of the current mesh.

We calculate new error quadrics at three points in the algorithms:

- In the preprocessing phase, as mentioned above, for each vertex we calculate the initial error quadric.

- When an intermediate vertex is created on an edge or a triangle, its quadric is calculated the same way: as a matrix sum of fundamental error quadrics of all faces, which are incident to the newly created intermediate vertex, divided by the order of the vertex and, optionally, weighted by their areas.

- After performing the operation we recalculate the quadrics of the newly created vertex and all adjacent vertices using the fundamental error quadric of the faces, which are currently incident to the respective vertices.

Before performing the operation, we calculate the *one-sided Hausdorff distance* d_v between all simplices of the original mesh, whose nearest points on the simplified mesh lie inside the neighbourhood $N_{s_1} \bigcup N_{s_2}$ of the contraction simplices s_1 and s_2, and the neighbourhood N_v of the newly created vertex v.

Here, if simplex s is a vertex, its neighbourhood N_s consists of the vertex itself and its incident faces and edges; if s is an edge, N_s consists of the edge and its incident faces; finally, if s is a face, N_s consists only of the face itself.

For two point sets A and B, the one-sided Hausdorff distance finds for each point in A the closest point in B and takes the maximum:

$$d(A,B) = \max_{a \in A} \min_{b \in B} \|a - b\|, \tag{5.1}$$

This function is not symmetric. The two-sided Hausdorff distance, or simply *Hausdorff distance*, is constructed to be symmetric by considering both of the one-sided Hausdorff distances and taking the maximum:

$$D(A,B) = \max(d(A,B), d(B,A)), \tag{5.2}$$

To ensure that due to certain special cases, the error introduced by a contraction operation does not exceed a predefined error threshold d_{max}, we compare d_v with d_{max}, and, if d_v exceeds d_{max}, we reject the operation.

Since the error quadrics are not accumulated, we use an approximation $\widetilde{d_s}$ of the Hausdorff distance to order the possible contraction operations in a priority queue:

$$\widetilde{d_s} = \max(d_{s_1}, d_{s_2}) + \sqrt{1/2 * e_q}, \tag{5.3}$$

where d_{s_1} and d_{s_2} are the one-sided Hausdorff distances from the neighbourhoods N_{s_1} and N_{s_2} of the simplices s_1 and s_2, respectively, to the original mesh.

The first part of this sum, $max(d_{s_1}, d_{s_2})$, is the Hausdorff distance to the original mesh of the neighbourhood $N_{s_1} \bigcup N_{s_2}$ of the contraction simplices before the operation. The quadric error e_q represents the sum of the squared distances from the newly created vertex to the incident faces of two contraction vertices. Therefore, the second part of the sum in equation 5.3, $\sqrt{1/2 * e_q}$,

is an approximation of the distance between the meshes before and after the operation.

Thus, the value $\widetilde{d_s}$ is a good estimation of the Hausdorff distance between simplified and original meshes. This makes this error metric compatible with Euclidian distances used to find the nearest simplices, as described in section 5.3.

5.2.2 Handling of boundaries and sharp features

In order to preserve boundaries we proceed as proposed by Garland and Heckbert. For each face incident to a boundary edge we generate a perpendicular plane running through the edge. Then we compute fundamental quadric for this constraint plane and add it to the quadric sums of both vertices incident to the boundary edge, divided by the order of appropriate vertex and optionally weighted by the squared length of the edge.

A further enhancement to the error quadrics is necessary to preserve sharp features. Figure 5.4 b shows the simplified helicopter of figure 5.4 a using the introduced error quadrics as discussed thus far. Sharp features as the propeller are destroyed. To handle very sharp edges properly we process them in the same way as boundaries.

As *feature edges* and *feature vertices* we define all edges and vertices, whose incident faces lie inside chosen small angle α_{max}. Note that according to this definition, boundary edges are also feature edges. For each detected feature edge e or feature vertex v we find the average plane of all its incident faces and compute its fundamental quadric. This fundamental quadric is then added to the quadric sums of both vertices incident to the feature edge or to the quadric sum of the feature vertex. Figure 5.4 c shows the helicopter simplified with feature preservation to the same number of vertices as in figure 5.4 b.

5.3 Spatial search data structure

In order to avoid a quadratic algorithm to pair close simplices for the contraction operations a spatial search data structure supporting nearest simplex queries is necessary. The data structure must be able to handle vertices, edges and triangles. Furthermore it must be dynamic as simplices are eliminated and sheared during the simplification process. We chose to use a regular grid for the spatial search because, as shown by Zachmann [109], it performs well in static and dynamic environments and is easy to implement. In each grid cell we stored a list of the simplices partially or completely contained in the cell.

In the beginning of the simplification process we used a grid with uniform edge length of twice the average edge length of the model, such that each simplex was in average contained in one or two cells allowing for fast insertion

a)

b)

c)

Figure 5.4: Simplification with and without feature preservation: a) original model (1972 vertices) with detected features drawn red; b) reduced to 200 vertices without feature preservation; c) reduced to 200 vertices with features preserved.

and removal. During the simplification process we kept track of the increasing average edge length and every time it exceeded the grid edge length, we destroyed the grid, created a new one of double grid edge length and inserted all remaining simplices to the new grid. In the case of models with low variance in the simplex size we achieved with this simple strategy, that in average each simplex had to be entered in only a constant number of grid cells.

5.3.1 Simplex insertion and removal

To insert a vertex we simply compute the enclosing cell and add the vertex to its list of contained simplices. Edges and triangles can penetrate more than one cell into which they had to be entered. In the case of an edge a simple incremental algorithm could be used to trace the penetrated cells along the edge. For triangles we implemented a not optimal but simple solution. First we collected all cells intersected by the bounding box of the triangle. Then we sorted out the actually penetrated cells by a marching cubes like strategy. Each of the grid vertices from intersected cells was classified to be *above*, *below* or *outside* of the triangle in consideration. Finally, the triangle was inserted in all grid cells with at least one vertex classified above and one classified below. As each triangle was in average inserted into a small number of cells the presented strategy worked reasonably fast.

For fast removal we stored for each vertex a pointer to the list item in the enclosing cell such that it could be removed in constant time. Similarly, we kept for each edge and each triangle a list of pointers to list items such that any simplex could be removed in time proportional to the number of penetrated cells.

Simplices incident to contraction vertices move in space during the contraction operation. It turned out that insertion and removal was so fast that it did not pay off to implement an optimized move operation for the regular grid. Instead we simply removed the simplex before the contraction operation and inserted it again afterwards.

5.3.2 Distance-sorted nearest neighbour queries

For our simplex pairing strategy described in the next section, it is necessary to find for each vertex besides the adjacent vertices, the closest non-incident simplex, and for each edge, the closest non-incident edge. We designed a general algorithmic scheme to find the closest simplex of a vertex or an edge, i.e. the *seed simplex*. The scheme exploits two priority queues, the *cell queue* to store the next to be considered grid cells sorted by increasing distance to the seed and the *simplex queue* to store the non-incident simplices from the considered cells also in increasing distance from the seed. The crucial idea is that the head of the simplex queue is the closest simplex to the seed, only if the head of the

cell queue, i.e. the closest not yet considered cell, is further apart from the seed. Now we can state the closest simplex search algorithm:

- Given a seed vertex or edge, initialize the simplex and cell queues to be empty.
- Lookup the *seed cells* penetrated by the seed, add all contained non-incident simplices to simplex queue and the cells adjacent to the seed cells to the cell queue.
- While the simplex queue is empty or the distance of the closest not considered cell is smaller than the closest simplex
  - extract the head of the cell queue, add all contained non-incident simplices to the simplex queue and add the not considered adjacent cells to the cell queue.
- Return the closest simplex from simplex queue.

For the closest simplex algorithm the following distance computations needed to be implemented: distance from vertex to vertex, edge, triangle or cell and the distance from edge to edge or cell. To avoid the square root operation we sorted the cells and simplices by the squared distance. As we were looking for the closest simplex, the distance computations from a vertex to an edge or triangle could be discarded if the orthogonal projection of the vertex did not fall inside the edge or triangle, because in this case there must be a vertex incident to the edge or an edge incident to the triangle closer to the seed vertex. The squared distance from a vertex at location p to an axis aligned box with lowest coordinates vector l and highest coordinates vector h can be computed by

$$dist = \sum_{\alpha \in \{x,y,z\}} \max\{0, l_\alpha - p_\alpha, p_\alpha - h_\alpha\}^2. \tag{5.4}$$

It is quite complicated to compute the distance from an edge to a cell. We used the following simple and efficient strategy. We parameterized the edge over $\lambda \in [0,1]$ as $p = o + \lambda \cdot v$, plugged this expression into equation 5.4 and minimized $dist(\lambda)$ over $[0,1]$.

Equation 5.4 was only evaluated on values where the selection of the max-function changed and on the extremes in-between. As the distance from a point moving on a straight line to a convex object can only assume one minimum in a connected region, it was sufficient to follow λ until it increased.

5.4 Description of the algorithm

Our simplification algorithm can be split into the initial preprocessing phase and the decimation loop itself. The algorithm proceeds according to an in-

creasing error, that is caused by contracting two simplices, which we call *correspondence pair*. All correspondence pairs are ordered in a priority queue.

5.4.1 Preprocessing

After the acceleration grid (see section 5.3) has been initialized, we identify the corresponding pairs for subsequent decimation operations.

For the priority queue algorithm to work properly, the operation causing the minimum error needs to be the first element in the queue. This can either be a manifold operation (edge collapse or edge flip) or a non-manifold one (vertex contraction or edge-edge contraction). As the error caused by manifold operations is not related to the distance by which the contraction vertices move, we need to consider all possible manifold-operations. In case of non-manifold operations, the caused error is at least half the distance between the contracted simplices. Therefore, we search the corresponding simplices of each vertex or edge with a maximum search distance of $2 * l$ and consider only the closest corresponding simplex. In case of the search for the corresponding simplex of a vertex, l is the length of the edge incident to the vertex that causes the minimum error if contracted. When we search for the corresponding edge of an edge, l is simply the length of the edge itself.

For each vertex v we proceed as follows:

- For all incident edges the error that would be introduced if the edge was collapsed is computed according to equation 5.3. We choose the edge with the minimum error ε_m and calculate its length l. Note, that the grid data structure automatically discards incident edges and triangles.
- Setting v as a seed vertex, we search in the grid with a maximum search distance $2 * l$ for the closest non-incident simplex (vertex, edge or triangle) using the algorithm described in section 5.3.2.
- If a simplex is found within the $2 * l$ distance, we compute the error ε_n, which will be introduced by contracting vertex v with the found simplex according to equation 5.3.
- We compare ε_m and ε_n and assign the operation with the minimal error $min(\varepsilon_m, \varepsilon_n)$ to the vertex v.

For edges only non-incident edges have to be considered as candidates. For each edge e_0 with length l we search the grid with a maximum search distance of $2 * l$ using e_0 as a seed edge for the algorithm described in section 5.3.2. During this search the algorithm guarantees that the points realizing the minimal distance between e_0 and the found edge e_1 are in the (open) interior of the two edges. For such two edges we compute the introduced error according to equation 5.3. Note, that we find the corresponding edge only for some edges.

After the search is complete, references to the found simplices are stored for all vertices and some edges. Vice versa, each simplex stores references to all vertices and edges pointing onto it.

Last but not least, the pairs are inserted into a priority queue according to their associated errors.

5.4.2 Decimation loop

At each step of the decimation loop we first take the pair with the minimal error from the queue.

Prior to perform the operation determined by this pair, the following tests are performed:

- *Normal test.* Here we check that the normals of triangles affected by the operation do not change by more than α_{max}.
- *Minimum angle test.* This test is performed only if the operation is an edge flip. We check if the minimum angle among all affected triangles increases by at least β_{min}. As described in section 5.1.1, this avoids endless loops while allowing to improve the shape of triangles.
- *Error test.* We calculate local Hausdorff distance from original to simplified mesh after performing the operation, and check if it doesn't exceed the global simplification threshold d_{max}, as described in section 5.2.
- *Collision test.* Optionally, the collision detection is performed, as described in section 7.3. This test allows to detect and avoid self-intersections which could occur as a result of the contraction operation.

When the operation is discarded because one of the above tests has failed, we put it into the second queue of *discarded operations*, otherwise we perform it.

After performing the operation we have to update all affected pairs. To explain this process, let's define V_c as the set of vertices that collapsed to a vertex v, E_c and T_c as the sets of edges and triangles changed by the operation, E_r and T_r as the sets of edges and triangles which were removed. We proceed as follows:

- For each vertex in V_c and each edge in $E_c \cup E_r$, we remove previous correspondences.
- We find correspondences for v and each edge in E_c as described above.
- For each vertex, which corresponds to an edge in $E_c \cup E_r$ or a triangle in $T_c \cup T_r$, we search again for the best correspondence.

- For each edge e, which corresponds to an edge from $E_c \cup E_r$, we find a new correspondence.

5.4.3 Double queue strategy

As described before, some pairs are discarded if one of the above tests fail. But it's not desirable to simply reject the operation. During simplification the neighbourhood of the considered pair might change and the reason of its rejection vanish. Therefore, we do not want to loose the operation.

On the other hand we cannot simply insert the discarded operation back into the priority queue, since its error is the smallest one and it would be taken from the queue again in the next step causing an infinite loop.

One strategy to keep invalid operations is to assign the discarded operation to all simplices that caused the invalidity. If a simplex is removed or changed in a contraction operation, all of the discarded operations attached to it are reconsidered. Unfortunately, this strategy demands for complex data structures and a large amount of computations. Therefore, we used a simpler heuristic strategy.

We put each discarded operation into a second queue of *discarded operations*. We considered one randomly selected simplex from this queue once in k simplification steps and after establishing a new correspondence for it we reinserted it into the first queue. We did it in the way to ensure that the frequency of reinsertion of discarded elements into the first queue is approximately proportional to the size of the second queue. When the first element was put into the second queue we initialized k by the number of elements in the first queue and decremented it by one in each simplification step. Each time a further element was put into the second queue k was updated as follows: $k := k * \frac{N_2}{N_2+1}$, where N_2 is the number of elements in the second queue. If k became zero it was set to $\frac{N_1}{N_2}$, where N_1 is the number of elements in the first queue.

5.5 Preservation of normals and other surface properties

To preserve the appearance of the object during simplification, we extend the geometric Hausdorff error measure with respect to appearance attributes as proposed by Klein et al. [56] for view-dependent multi-resolution meshes. However, in contrast to this approach we need an error measure that is independent of the viewing position.

When an edge is removed due to a collapse operation, the appearance attributes of the removed points are interpolated during rendering. A screen space error can now be defined as the distance between a shaded point of the

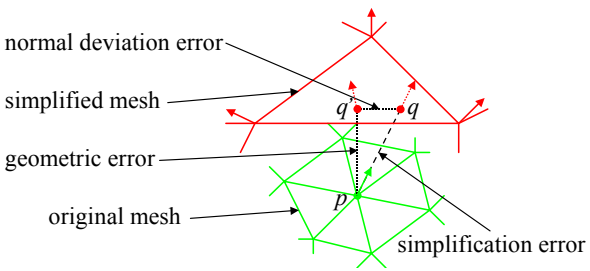


Figure 5.5: Combination of error measures for appearance preservation.

original model projected into screen space and the next pixel on screen with the same colour. For static LODs this distance can directly be transformed into object space as the distance between a point on the approximated surface and the next point on the original mesh with the same appearance attribute.

We define the simplification error in object space to be the distance of a point p on the original mesh and the closest point on the simplified mesh with the same interpolated normal q (see figure 5.5). Now we make the observation that the vector between the original point p and q can be split into the orthogonal vectors pq' and $q'q$, where q' is the closest point on the simplified mesh. Therefore, the simplification error ε can be written as a combination of the geometric Hausdorff error $\varepsilon_g eo$ and the normal deviation error on the simplified mesh $\varepsilon_a pp$:

$$\varepsilon^2 = \varepsilon_g eo^2 + \varepsilon_a pp^2 \tag{5.5}$$

The normal deviation error $\varepsilon_a pp$ can be approximated using the maximum normal curvature κ_1:

$$\varepsilon_a pp \approx \frac{\arccos(\vec{n} \cdot \vec{n}_i nt)}{\kappa_1}, \tag{5.6}$$

where $\vec{n}_i nt$ is the interpolated normal at q'. The maximum curvature of a point on a bilinearly interpolated triangular patch with specified per-vertex normals can be approximated by:

$$\kappa_1 \approx \max\left(\frac{\arccos(\vec{n}_1 \cdot \vec{n}_2)}{\|P_1 - P_2\|}, \frac{\arccos(\vec{n}_1 \cdot \vec{n}_3)}{\|P_1 - P_3\|}, \frac{\arccos(\vec{n}_2 \cdot \vec{n}_3)}{\|P_2 - P_3\|}\right) \tag{5.7}$$

For small angles, the computation of the inverse cosine can be saved, since in this case $\arccos(\vec{n}_a \cdot \vec{n}_b) \approx \|\vec{n}_a - \vec{n}_b\|$.

To prevent aliasing artifacts in the shading, we smooth the normals of vertices that are only adjacent to triangles smaller than $e_n ode/res$ before simplification. This also leads to a more efficient simplification.

While we only use normals in our examples, the described approach is able to deal with arbitrary appearance attributes, for which a distance is defined, e.g. per-vertex colours, BRDFs, etc.

5.6 Results

The first example in this section demonstrates the model of a steering wheel. A lot of artefacts, resulting from improper tessellation of a trimmed NURBS surfaces, are present in this model. Figures 5.6 a – 5.6 d show several steps of a simplification algorithm, which performs only vertex contractions.

Many gaps between separate patches, hardly recognizable in the original model, have not been sewn together. Some of them have been transformed to real holes and remain even in a coarse model with only 125 vertices. In contrary, our new simplification algorithm which performs also generalized pair contractions allows to close such narrow gaps already at the early stages as shown in figures 5.6 e – 5.6 h. In a model reduced to 500 vertices all artefacts have been eliminated, and only the large holes, which are inherent to the model; remain.

Figures 5.7 show the model of a microscope. The original model in figure 5.7 a looks perfect but some triangles in the upper part of the tube are missing. Figure 5.7 b depicts the model simplified with vertex contractions only. As can be seen, after simplification to 250 vertices the holes in the tube not only remain but have increased. Furthermore, the mirror at the bottom originally consisting of several independent parts becomes strongly corrupted. The model in figure 5.7 c was simplified with generalized pair contractions. As expected, the holes in the tube were closed and also the mirror is simplified adequately.

5.7 Related work

Since simplification is one of the fundamental operations performed on polygonal meshes, there is an extensive amount of different methods. However, there are detailed reviews of available simplification algorithms, e.g. by Cignoni et al. [17] or by Luebke [64], and we will give here only a short overview of the most related approaches.

Mesh simplification. The family of *vertex clustering* methods has been introduced by Rossignac and Borrel [84] and has been refined in numerous more recent works, e.g. by Low and Tan [62]. The algorithms of this family essentially proceed by applying a 3D grid to the object and for each cell contracting all the vertices inside the cell. Although the degenerate faces are subsequently

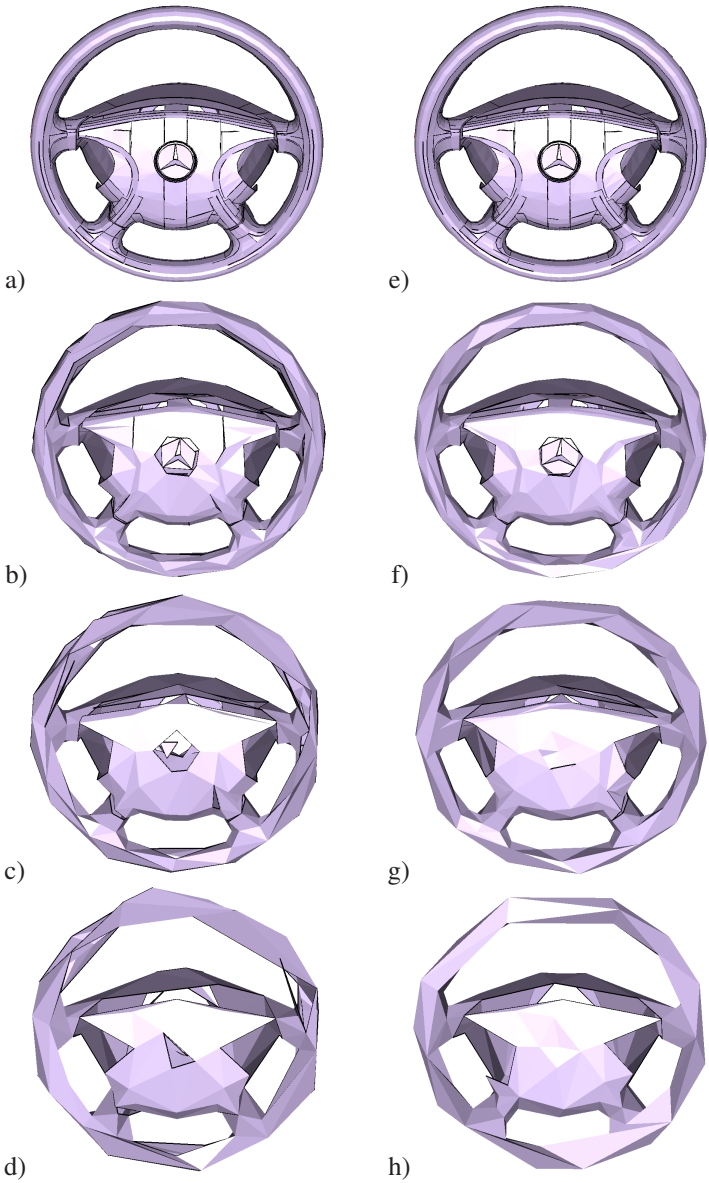


Figure 5.6: Several stages of simplification with only vertex contractions (a – d) and with generalized pair contractions (e – h): 4288 vertices (a and e), 500 vertices (b and f), 250 vertices (c and g), 125 vertices (d and h). While in the first case some gaps between patches remain even at the coarsest resolution, in the second case they all are closed at the early stage.

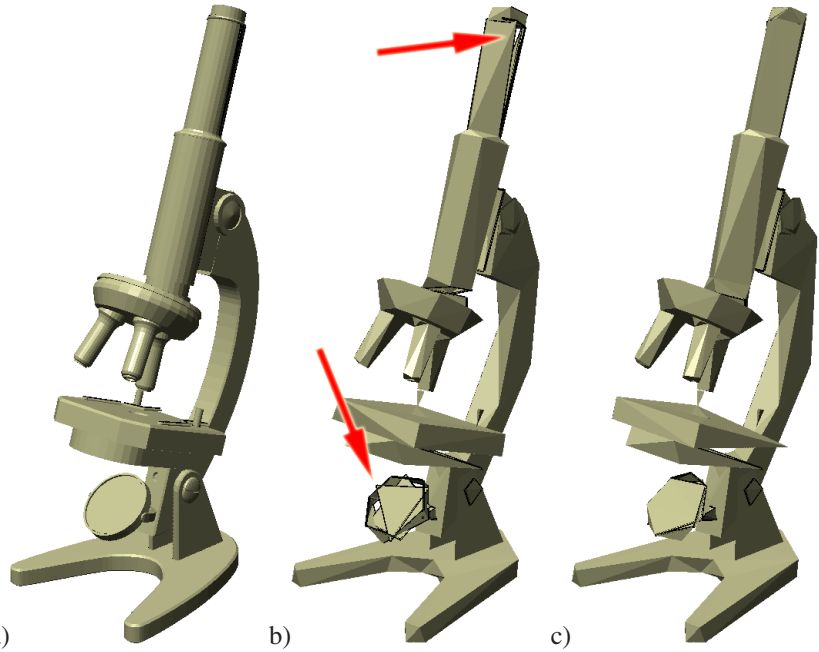


Figure 5.7: The original model of microscope with 4467 vertices (a) and simplified to 250 vertices with only vertex contractions (b) and with generalized pair contractions (c).

removed, it is difficult to influence the fidelity of the result due to lack of control over induced topological changes, and the reduction rate is quite low at flat regions of the model.

Cohen et al. [20] developed *simplification envelopes* to guarantee fidelity bounds while enforcing local and global topology. The simplification envelopes consist of two offset surfaces at some distance from the original surface. Since these envelopes are not allowed to self-intersect, this distance is decreased at high curvature regions. By keeping the simplified surface inside these envelopes, the algorithm can guarantee a geometric deviation of at most this distance and additionally it checks that the surface does not self-intersect. While this algorithm has the advantage to guarantee a geometric error bound, it is quite slow and requires an orientable manifold for the construction of the offset surfaces.

Zelinka and Garland [111] modified the above approach by using *permission grids* – spatial occupancy grids, where an operation is only performed if all cells that are intersected by the new triangles are allowed to be occupied. Although the algorithm is much faster than simplification envelopes and doesn't

need an orientable manifold mesh, the simplified models often contain much more triangles due to the discrete grid and the fact that the *Manhattan* distance is used instead of the *Euclidean*.

The vertex contraction operation, which was introduced at the same time by Popović and Hoppe [79] and Garland and Heckbert [29], has become the most common operation and is used in many simplification methods. In conjunction with the *quadric error metric* introduced by Garland and Heckbert [29], it offers flexible control over the quality, still at very high reduction speed. However, the quadric metric mostly overestimates the real geometric error which results in non-optimal reduction rates and the need to measure the exact error after simplification.

Appearance-preserving simplification. In the field of appearance-preserving simplification and rendering, the general approach is to use appearance-preserving level-of-detail. Garland and Heckbert modified their error quadrics [29] to preserve colour, texture coordinates and normals [30]. However, guaranteeing a certain error of the geometry or the appearance during rendering is not possible using these modified error quadrics. As a different error measure for appearance-preserving out-of-core simplification, the curvature of the mesh can be used, as in the paper by Lindstrom [60], but like for the modified error quadrics no screen space error can be guaranteed for this method.

Another approach used for view-dependent refinement of multi-resolution meshes was introduced by Klein et al. [56]. It is able to control the shading error by guaranteeing that for each point on the screen the distance to the next correctly shaded pixel is below a specified constant. Unfortunately, this method cannot be used for static LODs, since the error measure is viewpoint-dependent and requires the exact position and orientation of the surface on the screen to be known. Furthermore, the derivatives are calculated in screen space, which makes it unapplicable to precomputed static LODs.

A different approach is *perceptually driven simplification* proposed by Williams [104]. But again this method requires knowledge of all viewing parameters – even for its basic features that produce results similar to appearance-preserving simplification – and additional movement information for velocity simplification. Finally, peripheral simplification even requires tracking of the user's eye movements.

Measurement of simplification errors. For the first time, the Hausdorff distance, which is the tighest possible bound on the maximum distance between two surfaces (see, e.g., the book by Luebke et al. [63]), was used to control the simplification error by Klein et al. [55], although with significant computational effort.

Cignoni et al. [19] introduced the first method dedicated exclusively to measurement of errors on simplified surfaces, which allows to compare qual-

ity of different simplification methods. Another method, presented by Aspert et al. [3], is more efficient in terms of speed at the cost of higher memory use. Its implementation, the *Mesh* tool, was used in our research to compare the results of different simplification methods (see sections 6.2 and 7.5). Both algorithms are based on sampling of the geometry of the two models being compared, where the sampling density depends on the desired accuracy: in order to double the accuracy the number of samples needs to be multiplied by four. Therefore, these algorithms quickly become slow for higher accuracy.

5.8 Summary

We presented an important strategy to generate high-quality simplified models. We introduced the generalized pair contraction operations. They not only allow to remove gaps and holes, but also seamlessly integrate the automatic connection of close surface parts in the most general setting and the resolution of initial self-intersections during the simplification process.

Chapter 6

Out-of-core simplification

Modern 3D acquisition and modelling tools generate high-quality, detailed geometric models, and the associated complexity increases much faster than the hardware performance. Digitizing human-size objects in the sub-millimeter range has become common, posing new challenges for the processing and rendering of this data. Two standard examples of such hufe models are shown in figures 6.1.

Whereas earlier simplification algorithms have worked only with models that completely fit into main memory, the necessity of methods, which can deal with arbitrary large meshes, has become obvious. These out-of-core algorithms do not load the whole geometry of a model into the actual in-core memory, but temporarily store its large parts on disk. Therefore, the memory requirements of these methods is independent of the complexity of the input as well as output models.

The fact that the model cannot be loaded into memory prevents efficient comparison of simplified and original objects, which in turn complicates the control over the geometric error of the simplified mesh. As long as the resulting model does not fit into main memory as well, error control is simply impossible in most cases.

Applying the methods mentioned above, we developed a high-quality end-to-end out-of-core mesh simplification algorithm (neither input, nor output model fit into main memory), which is capable not only to measure the Hausdorff distance between the original and simplified meshes, but also to simplify a model up to a given error threshold. It guarantees, that no operation is performed what would exceed this threshold, which allows to get a very high reduction of the model complexity at a certain maximum geometric error.

Furthermore, applying generalized pair contractions instead of only vertex contractions allows for a controlled modifications of the topology. This way small gaps are automatically sewed and parts which are close together are merged in a controlled way during the simplification process.

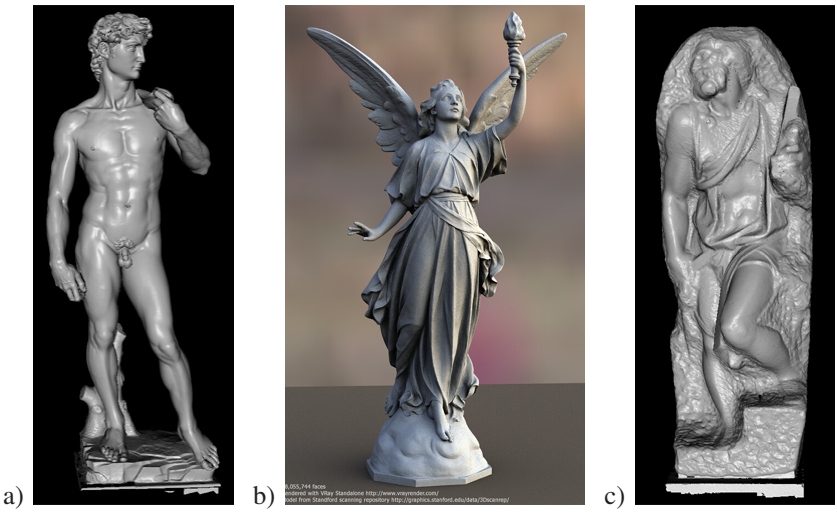


Figure 6.1: Three standard out-of-core models: a) David with 56 230 343 triangles; b) Lucy with 28 055 742 triangles; c) St. Matthew with 372 422 615 triangles.

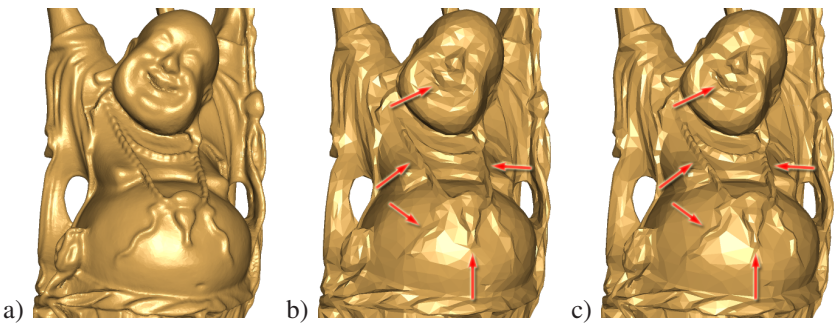


Figure 6.2: a) Original Happy Buddha model with 1 087 716 vertices; b) reduced to 18 338 using standard QEM simplification; c) reduced to 18 338 vertices using our out-of-core simplification method. Their corresponding relative Hausdorff maximum errors (over the bounding box diagonal) are 0.79% and 0.26%, respectively.

Model	Δ_{input}	Δ_{output}	$d = 7$	$d = 9$	$d = 11$	$d = 13$
Armadillo	345 944	33 780	737	1 022	n.a.	n.a.
Happy Buddha	1 087 716	32 377	1 336	488	1 022	n.a.
David (2 mm)	8 254 150	25 888	2 637	2 029	826	n.a.
Lucy	28 055 742	26 772	1 551	1 008	430	1 024

Table 6.1: Numbers of triangles before and after simplification and maximum numbers of triangles in the simplified nodes at different levels of the octree hierarchy (with depth 7, 9, 11 and 13) for four standard models used in our test.

The amount of main memory required for our algorithm does not depend on the size of the input or output models and can easily be configured to consume a fixed amount of memory depending on the system it is running on.

Of course, these advantages lead to lower computation rates compared to other recent out-of-core simplification methods.

This chapter is structured as follows. First, we describe our out-of-core simplification algorithm in detail. Some results of its work are shown in section 6.2. Finally, we discuss the related work and conclude.

6.1 Description of the algorithm

Since generalized pair contractions close gaps more efficiently than vertex contractions, a simple and fast out-of-core simplification is possible by cutting the model into subparts and simplifying each subpart independently. With generalized pair contractions, gaps are automatically closed, when all subparts are simplified together. To simplify gigabyte-sized models, partitioning and independent hierarchical simplification are applied recursively.

During each node simplification a maximum geometric error threshold for the node simplification is determined as a constant fraction of the edge length of its bounding box. Therefore, the error threshold duplicates with each level of the octree. This leads to an almost constant order of magnitude of triangles in the simplified node[1], as shown in table 6.1. The geometric approximation error of the simplified model is measured against two levels below the current node. This way, only the geometry of at most 64 nodes have to be loaded into memory. Nevertheless, a good upper bound for the geometric deviation from the original model can be guaranteed.

If the accumulated maximum error in the next level already exceeds the given global error threshold, the nodes are simplified up to this error instead

[1] Of course this number depends on the fractal dimension of the underlying mesh. But most of the meshes have a fractal dimension near 2, which is verified by our experiments.

and the hierarchical simplification is stopped. In this way all nodes are simplified up to the desired error. Since the simplification of nodes in the same level of the hierarchy is completely independent of each other, it can be parallelized in a straightforward way by distributing the nodes to simplify between different computers.

To combine the subparts into one connected model, we use two different approaches, depending on the size of the final simplified model. In general, during simplification all 64 grandchildren of a node are gathered into the current node and simplified. During simplification, the introduced gaps along the cutting planes between them are automatically closed, since we know that their geometric distance is at most half of the approximation error threshold of the current node. Therefore, if possible, we do not perform independent simplification of the subparts on the last level of the hierarchy, but in-core simplification of the combined model. When end-to-end out-of-core simplification is required, we perform an out-of-core stitching of the subparts after the last level of the hierarchy is simplified.

The following sections describe each phase of our algorithm in detail.

6.1.1 Cutting

Since the gaps are automatically closed during hierarchical simplification, we do not need to preserve the triangles at node boundaries (in contrast to the method by Cignoni et al. [18]). But if the triangles are simply sorted into one of the child nodes during partitioning, a sawtooth boundary is created, which cannot be simplified efficiently without exceeding the given error tolerance of the node along the boundary. Therefore, if the model contains more than T_{max} trianglesthe, it is partitioned by cutting the geometry of a node into eight subparts and storing it in its children. The partitioning is repeated until no node has to be cut anymore. If no geometry is contained in a node, it is marked and not partitioned further. In this way a sparse octree is build.

Since the whole geometry of a node and all its children generally does not fit into the main memory, the vertices and normals of the mesh are stored in blocks and swapped in and out from disk using a last-recently-used (LRU) algorithm. The indices of the triangles need not to be stored in memory and therefore, can be streamed from the geometry file of the node to the files of its children. This is accomplished by loading the current triangle from the geometry file of the node, cutting it and then saving the generated triangles in the child geometry files. Therefore, only the current triangle and the triangles generated from it are stored in memory. After the triangle is cut, it is not needed any more. When first saving all triangles in the root node, the vertex normals are calculated.

At each partitioning step every triangle is cut with the three planes dividing the node into its children and the resulting triangles are stored in the appropriate geometry files. When a triangle edge is cut, the normal of the new point is calculated by linear interpolation. Note that new vertices may have the same coordinates as existing vertices, but this is resolved when the whole tree is build. After partitioning the triangles of a node and storing it in its children, the geometry file of this node is not used any more and is deleted.

When the partitioning is complete new indices for the leaf node triangles are calculated and duplicate points are removed.

The total complexity of the partitioning algorithm is $O(n \log n)$, since on each level of the octree all triangles need to be processed once.

6.1.2 Hierarchical simplification

After partitioning, the geometry contained in the leafs of the octree is stored on disk. Starting from the geometry of these nodes the model is simplified recursively from bottom to top with a constant resolution *res* depending on the node size using the following algorithm:

- At every level of the octree, the simplified geometry from all child nodes that are two levels below the current node (or the original geometry if there is no pre-simplified geometry at this depth) is gathered. Its approximation error ε_{prev} is then the maximum error of the simplified geometry in these child nodes or zero.
- The resulting geometry is simplified as long as the Hausdorff distance ε_h to the gathered geometry is less than $\varepsilon_s = \frac{e_{node}}{res} - \varepsilon_{prev}$, where e_{node} is the edge length of the current nodes bounding cube and *res* is the desired resolution in fractions of e_{node}.
- $\varepsilon = \varepsilon_h + \varepsilon_{prev}$ is stored as approximation error in the current node.

By using the children at two levels below the current node instead of its direct children the simplified geometry contains less triangles, since the approximation of the real geometric error is better. This is due to the fact that the difference between the estimated geometric error ε and the real geometric error ε_{real} is low, since:

$$\varepsilon_{real} \geq \varepsilon_s = \frac{e_{node}}{res} - \varepsilon_{prev} \geq \frac{e_{node}}{res} - \frac{e_{node}}{4 \cdot res} = \frac{3}{4}\frac{e_{node}}{res} = \frac{3}{4}\varepsilon \tag{6.1}$$

and thus $\frac{3}{4}\varepsilon \leq \varepsilon_{real} \leq \varepsilon$.

Starting with the already simplified geometry gathered from the grandchildren of the current node greatly reduces the computation cost and still leads to high-quality drastic simplifications. Since the input and output number of

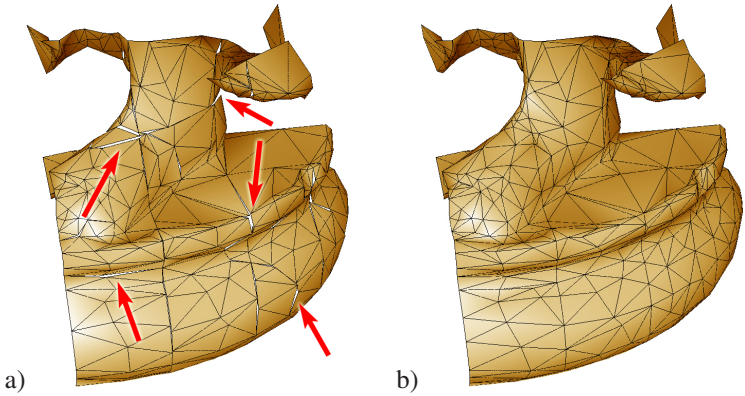


Figure 6.3: Hierarchical simplification using only vertex contractions (a) and generalized pair contractions (b). The arrows point to some of the cracks introduced by cutting and independent simplification and not closed by vertex contractions.

triangles in an octree cell generally remain in the same order of magnitude and since the vertices inside a node not closer than $\varepsilon = \frac{e_{node}}{res}$ are bound to be less than $\frac{12}{\pi} res^3$, the complexity of the simplification algorithm linearly depends on the number of nodes in the octree and, therefore, is $O(n)$. This means that the total simplification time depends only linearly on the number of leaf nodes and thus linearly on the number of triangles in the base geometry. Therefore, the total time for this out-of-core simplification algorithm sums up to $O(n \log n)$, where n is a number of input triangles.

In order to close the cracks introduced by the cutting and independent simplification in previous stages of the recursion, the simplifier has to be capable of performing topological simplification. Performing standard vertex-contraction-based simplification on such data could have undesirable results, as shown in figure 6.3 a.

Therefore, the generalized pair contractions operator described in section 5.1 is used. The use of this technique eliminates the cracks introduced by the cutting and independent simplification in previous stages of the recursion by automatically sewing disconnected parts together, as shown in figure 6.3 b.

6.1.3 Stochastic simplification

As a criterion for the choice of next contraction operation we use the quadric error metric presented by Garland and Heckbert [29].

Although the quadric error metric is a fast technique, which provides good results, it does not deliver the Hausdorff distance. In our case this is a neces-

sary requirement. Therefore, in addition to quadric error metric we calculate the Hausdorff distance between the original and the simplified meshes. It is done the same way as described by Hoppe [45] and Klein et al. [55]. Before contracting the chosen candidate pair we always check, if the Hausdorff error which will be produced by this operation is less than the given error threshold. If not, we reject the operation. Thus we avoid all operations whose errors exceed the maximum error set for the given hierarchical level.

During each node simplification an idea proposed by Wu and Kobbelt [107] is used. Instead of using a priority queue to order candidates for contraction operations, at each simplification step we stochastically pick N_{rand} vertices v_i – candidates for the next contraction operation. Then, for each candidate vertex the neighbour simplex s_i is found, such that contraction of v_i and s_i will result in the smallest quadric error. In section 5.3.2 this search procedure is described in detail. Since the search of nearest neighbour simplices is expansive, we do it for N_{search} vertices only. For the rest $N_{rand} - N_{search}$ vertices we check only their adjacent vertices (this means that for these vertices only edge collapses could be found). Of course for vertices which lie on boundaries, in order to close the cracks introduced by the cutting, we always have to perform the complete search[2].

After defining N_{rand} candidate contraction pairs, we choose the one with the smallest quadric error that will arise after contracting it. The new position of a contraction vertex is chosen in order to minimize this error.

Once an operation is rejected we mark the vertex with a flag, which is valid only until the operation on a neighbour simplex is performed. If a randomly chosen vertex is marked with this flag, we choose the next vertex. Once all operation candidates have been rejected and marked, the simplification of a given node could not be continued further without exceeding the maximum error threshold and we stop.

Table 6.2 demonstrates how quality and performance rates of our algorithm depend on the number N_{rand} of the vertices, randomly selected at each simplification step. Computations have been done for the Armadillo model, shown in figure 6.4, with an error threshold set to 0.129% of the diagonal of the bounding box. For all other models the results are similar. In the last row of the table the rates for the similar simplification algorithm driven by a priority queue are shown. In shorter times the stochastic approach achieves even greater reduction rates than using a priority queue. Note, that all times include the cutting time ($\approx$1:10), which does not depend on simplification parameters.

[2] In practice, we performed the complete search of nearest neighbour simplices only for boundary vertices.

Figure 6.4: The Armadillo model, originally containing 345 944 triangles, simplified to 33 780 triangles.

N_{rand}	Δ_{output}	Time (m:ss)	Rate (Δ/sec)
4	33 780	**6:18**	**826**
6	**33 733**	6:35	790
8	33 775	6:47	767
10	33 933	7:15	717
Queue	33 829	8:56	582

Table 6.2: Impact of the number N_{rand} of the vertices, randomly selected at each simplification step, on the reduction and performance rates for the Armadillo model.

6.1.4 Stitching

To generate a consistent mesh from the independently simplified nodes we move a stitching frame over the model. This frame is placed as shown in figure 6.5. For all border vertices inside this frame the closest simplex in the other seven nodes is determined and a contraction operation is applied if the distance is less than 2ε. In this way all gaps introduced by independent simplification of the nodes are closed.

Finally duplicate vertices are removed and new global indices stored in each node. In this way a new vertex index can be calculated by only checking the direct neighbour nodes leading to a stitching time of $O(n)$, where n is the number of input triangles. Then the simplified and stitched geometry is written into a single file that may again exceed the amount of main memory available.

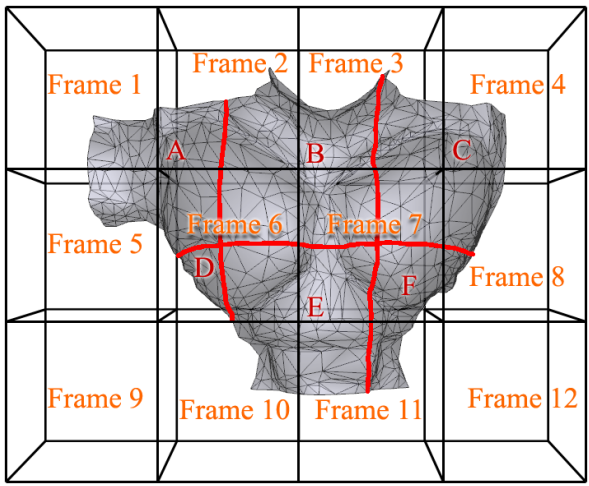


Figure 6.5: Stitching frames for the torso of the Armadillo model.

a) b)

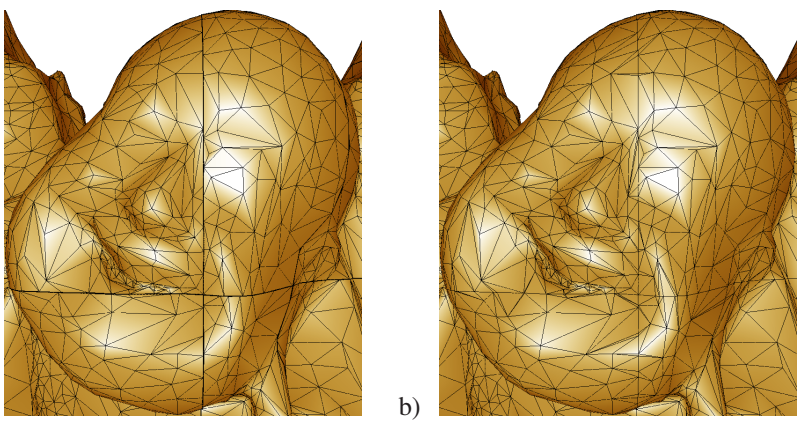


Figure 6.6: The head of the Happy Buddha model before (a) and after (b) stitching.

Model	Error (% of diagonal)	Cutting time (h:mm:ss)	Simpl. time (h:mm:ss)	Rate (Δ/sec)
Armadillo	0.129	0:01:12	0:05:06	826
Happy Buddha	0.170	0:04:40	0:19:28	728
David (2 mm)	0.178	0:38:01	2:22:02	762
Lucy	0.163	2:19:08	8:03:57	779

Table 6.3: Reduction and performance rates of our out-of-core simplification algorithm running on a single PC for four standard models.

6.2 Results

All results presented here have been measured on a Pentium-4 processor with 1.8 GHz and 512 MB main memory. Like other methods, we restrict ourselves during the simplification to the one-sided Hausdorff distance from the simplified to the original model.

In table 6.3, the reduction and performance rates of our algorithm for four models from the Stanford 3D Scanning Repository[3] and The Digital Michelangelo Project [4] are shown. The Lucy and David models are shown in figure 6.1.

The simplification time for these models is split into three parts. The cutting of the model has an approximate splitting rate of 25 000 / $\log n$ triangles/sec, where n is a number of input triangles, and simplification algorithm itself has an approximate reduction rate of 960 triangles/sec. The stitching algorithm was not applied, since the simplified models fit into main memory, but it performs at more than 100 000 triangles/sec. Since the hierarchical simplification can be parallelized, we ran the simplification on ten PCs achieving a linear speedup of the reduction rate by a factor of ten (see Guthe et al. [9]).

We compared quality of our algorihnm with the following previous simplification methods:

1. QEM simplification by Garland and Heckbert [29] (QSlim 2.0), the only in-core method used in our comparison;
2. simplification using *out-of-core clustering* by Lindstrom [59] (OOCC);
3. out-of-core simplification using *external memory management* by Cignoni et al. [18] (OEMM-QEM);
4. *stream decimation* by Wu and Kobbelt [107].

Simplification errors for the Happy Buddha model (initially containing 1 087 716 triangles) were measured using the *Mesh* tool developed by Aspert

[3] http://www-graphics.stanford.edu/data/3Dscanrep
[4] http://www-graphics.stanford.edu/projects/mich/

Method	Δ_{output}	One-sided Hausdorff error (% of diagonal)	Symmetric Hausdorff error (% of diagonal)
QSlim v2.0	18 338	0.261	0.786
OOCC	19 071	0.919	0.919
OEMM-QEM	18 338	0.505	0.821
Stream decimation	18 486	0.488	0.818
Our method	18 248	0.176	0.706

Table 6.4: Results of applying different out-of-core simplification methods to the Happy Buddha model.

et al. [3]. As table 6.4 demonstrates, both one-sided and symmetric Hausdorff distances between simplified and original meshes in our approach are smaller even than in in-core QSlim. Of course, since we use the one-sided Hausdorff distance during simplification, it is significantly lower than the symmetric (double-sided) Hausdorff distance.

In figures 6.7 and 6.8 it is clearly visible that, compared to the other methods, details, e.g. the necklace or the mouth, and silhouettes are much better preserved by our algorithm. The very small error of our method is a result of rejection of contraction operations, which would introduce a too large Hausdorff error (see section 5.2). Since in out method error quadrics are not accumulated, we achieve much better reduction rates in flat regions with noise, as figures 6.8 g – i clearly demonstrate.

6.3 Related-work

To simplify models of ever increasing size a number of out-of-core simplification algorithms have been developed. El-Sana and Chiang [25] sort all edges according to their length and use this ordering as decimation sequence. In a more efficient algorithm by Lindstrom [59], *vertex clustering* is used to reduce the number of vertices; but since the geometry data is stored in a voxel grid the memory requirement of this algorithm depends on the output size of the model. For cases where neither input nor output model fit into main memory, Lindstrom and Silva developed an *out-of-core vertex clustering* method [61]. The multiphase algorithm by Garland and Shaffer [31] first uses vertex clustering to reduce the complexity of the input model and then greedy simplification approach to achieve high-quality results.

Another general strategy for out-of-core simplification is to split the model into smaller blocks, simplify these blocks and stitch them together for further simplification. Hoppe [46] applied this approach to terrain, while Erikson and

Figure 6.7: Results of applying different simplification methods to the Happy Buddha model: a) original model (1 087 716 triangles); b) QSlim 2.0 (18 338 triangles); c) OOCC (18 338 triangles); d) OEMM-QEM (18 338 triangles); e) stream decimation (18 486 triangles); f) our method (18 248 triangles).

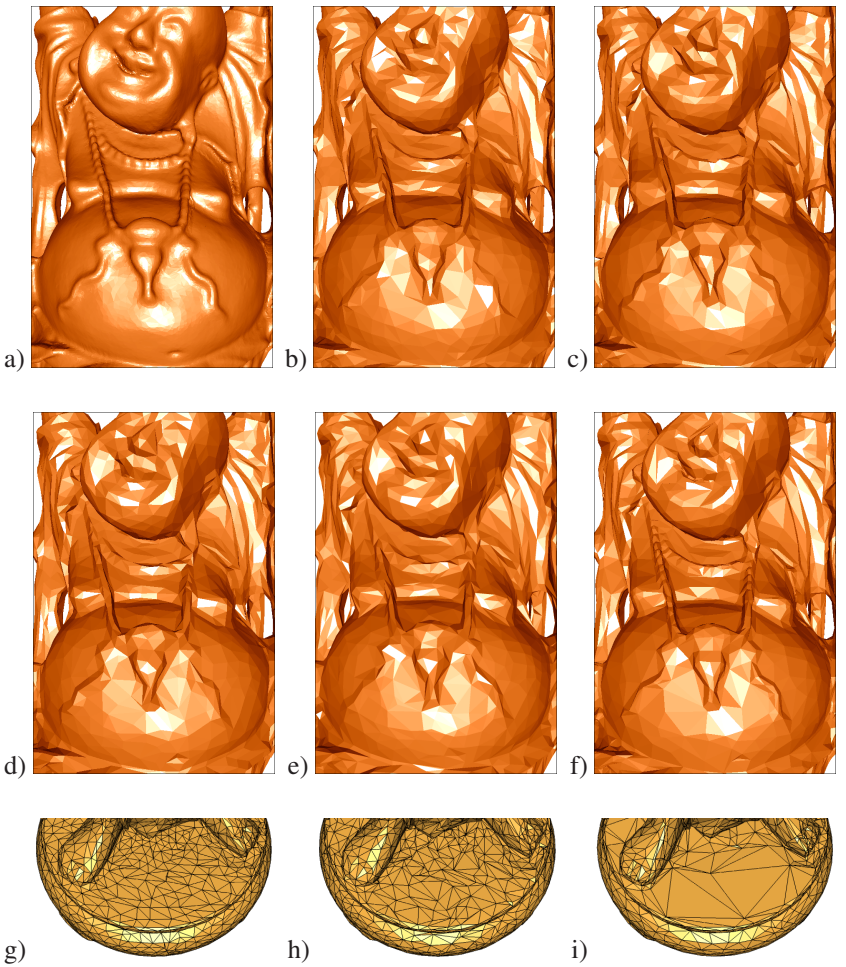


Figure 6.8: Results of applying different simplification methods to the Happy Buddha model – close-ups: a) original model (1 087 716 triangles); b) QSlim 2.0 (18 338 triangles); c) OOCC (18 338 triangles); d) OEMM-QEM (18 338 triangles); e) stream decimation (18 486 triangles); f) our method (18 248 triangles); g) OEMM-QEM; h) stream decimation; i) our method.

Manocha [26] and Prince [80], to arbitrary meshes. This approach has the problem that triangles intersecting the octree cells used to partition the model cannot be simplified before the cells are combined in a higher level of the hierarchy. Therefore, the number of triangles in an octree cell may very well exceed the main memory available, so these are not real out-of-core simplification algorithms although they allow simplification of large models. To overcome this problem a special method to simplify these border triangles has been developed by Cignoni et al. [18]. In our work we showed that the generalized pair contractions combined with cutting of the model at octree cell boundaries provide a more elegant and general solution to this problem.

Wu and Kobbelt developed a *stream decimation* algorithm [107] for out-of-core simplification, which performs decimation by collapsing randomly chosen edges. However, the geometric distance between the original and simplified models cannot be truly controlled, since the original model in the active working region does not fit into main memory. Also the problem may arise that the currently processed triangles may not fit into main memory.

Isenburg et al. [49] proposed a *processing sequence* paradigm, which represents a mesh as a particular interleaved ordering of indexed triangles and vertices, a representation that allows streaming very large meshes through main memory while maintaining information about the visitation status of edges and vertices. They apply this approach to out-of-core simplification [50], as well as to other mesh processing tasks, such as remeshing, parameterization or smoothing.

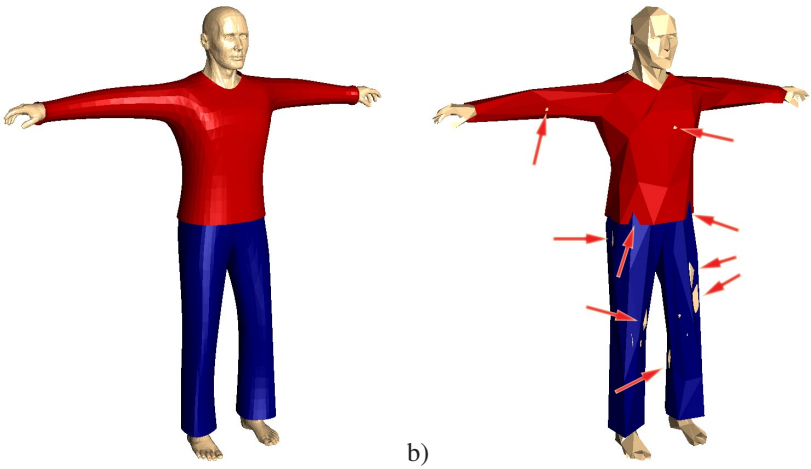
6.4 Summary

We presented a high-quality end-to-end out-of-core mesh simplification algorithm. The main features of the algorithm are that it allows to guarantee a maximum geometric distance between original and simplified models and that topological simplification is controlled based on a geometric error. Furthermore, the maximum allocated main memory can be restricted by the user. Although, due to the advantages of the algorithm the reduction rates are less than of other recent algorithms, they are almost constant regardless of the size of the input model. This demonstrates the optimality of the approach.

Chapter 7

Intersection-free simplification

Currently, a large number of simplification algorithms exist that produce high-quality approximations of complex models with a reasonable amount of polygons. However, none of the known techniques avoids the generation of self-intersections during the simplification process. Figures 7.1 and 7.2 show typical examples, where self-intersections lead to severe visual artefacts.



a) b)

Figure 7.1: Model of a man wearing a shirt and trousers with 17 060 vertices (a) was simplified to 600 vertices, arrows show self-intersections (b).

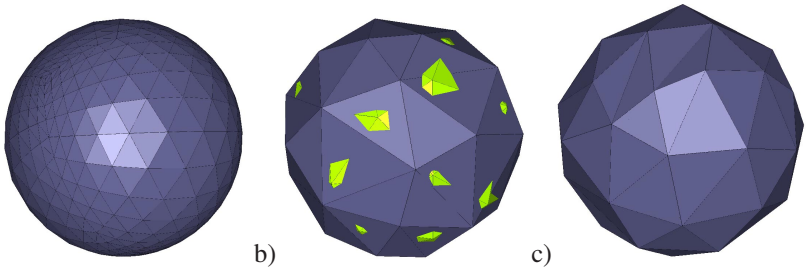


Figure 7.2: Model consisting of a green sphere inside a blue sphere (a) was simplified to 70 vertices, without (b) and with (c) avoidance of self-intersections.

In figure 7.1 a, a human body is dressed with a shirt and trousers. A realistic adaption of the clothes to the body typically demands for a computationally expensive physical simulation with a not too coarse resolution of the model. For an efficient visualization, the resolution of the model should be adaptable to the extent of the model in screen space, what can be achieved by the use of LODs. A standard simplification algorithm generates coarse approximations of the model with artifacts, as shown in figure 7.1 b, where the combined model was reduced to 600 vertices. The body penetrates the cloth in several places. A simulation of the cloth in such a coarse resolution is not feasible and, therefore, is a simplification algorithm that avoids self-intersections is mandatory.

The second, purely synthetic model, shown in figure 7.2 a, is composed of two nested spheres. After reduction to 70 vertices, this model also contains multiple self-intersections, as shown in figure 7.2 b.

Self-intersection problems typically arise, when two or more close surface layers are present, what is very common for models of dressed humans, for which our method was originally designed. We simplified the man and the nested spheres once with edge collapses and once with vertex contractions, where the contracted vertices do not have to be adjacent. Both cases yielded results similar to figures 7.1 b and 7.2 b.

We developed a method that allows to completely avoid self-intersections without loss of approximation quality. We describe our self-intersection-free simplification strategy based on the vertex contraction approach, as introduced by Popović and Hoppe [79] or Garland and Heckbert [29]. The result of simplification of the nested spheres model using our approach to avoid self-intersections is shown in figure 7.2 c.

The proposed approach have similarity with the simplification strategy of progressive simplicial complexes [79]. The atomic simplification operation is the vertex contraction. Starting with the original fine mesh vertex by vertex is removed via pair contraction operations. At any time we keep the set of all vertex pairs suited for contraction. This set is called the *potential set* and

includes all vertex pairs connected by an edge, the so called *edge pairs*, and for each vertex the closest not adjacent vertex forming the *non-edge pairs*.

The non-edge pairs are determined with the spatial search data structure described in section 7.1. To allow for fast simplification we use the quadric error metrics introduced by Garland and Heckbert [29] with their extension to border edges.

For a self-intersection-free simplification scheme it does not make sense to start off with a model that has self-intersections. Therefore, we first clean up the model to eliminate all initial self-intersections, as described in section 7.2.

In the initialization of the simplification process the vertex pairs of the potential set are inserted into a priority queue as introduced by Klein et al. [55], which is sorted by the quadric error produced if the vertex pair would be contracted. Step by step the vertex pair producing the smallest error is extracted from the queue.

Before the pair contraction is performed we check if the normals of the incident triangles flip, as suggested by Ronfard and Rossignac [82], and, if the normal check succeeds, we check if the pair contraction produces a collision leading to a self-intersection, as described in section 7.3. If a collision is detected, we try to avoid it by determining a different target position, as described in 7.4.

If a collision-free position is found we evaluate the quadric error metric at the new target location and re-insert the operation into the priority queue. If the collision avoidance fails, we add the operation to a second *queue of discarded operations* described below and, which gathers operations that failed in the normal check.

If both normal and self-intersection tests succeed, the operation is performed. The quadric error metric of all potential vertex pairs containing an affected vertex is re-computed and their priorities are updated in the priority queue. For each vertex pair containing an affected vertex and not connected by an edge we re-compute the closest non-adjacent vertex for the non-contracted vertex and add the new pairs to the priority queue.

Once discarded operations can become valid after contraction operations in the neighbourhood that remove the cause of invalidity. Book marking the operations that invalidate a temporarily discarded one can become very complicated. Therefore, we chose to use the strategy with the second queue gathering all invalid operations. We re-consider the invalid operations with a randomized strategy. With a frequency proportional to the number of operations in the queue of discarded operations we randomly select a discarded operation, find the currently best correspondence for it and enter it into the priority queue.

The self-intersection test is not performed before the insertion of a pair into the queue because it is more expensive than the computation of the quadric error metric. In this way we have to compute only one collision test per collapsed

vertex or avoided collision instead of one per insertion of a vertex pair into the queue, which is in the average case of manifold meshes (see definition 2.6 from section 2.1.1.1) eight times the number of vertices and for non-manifold meshes potentially even more.

7.1 Dynamic spatial search data structure

At three places of our intersection-free simplification method we make use of a spatial search data structure:

1. to remove initial self-intersections, all pairs of an edge and a triangle whose intersection need to be found;

2. to pair close vertices for non-edge vertex pairs, the closest non-adjacent vertex of another vertex has to be found;

3. to detect collisions during vertex contraction, all simplices in the vicinity of the contracted vertices need to be known.

As simplices change their shape and location during simplification, the spatial search data structure needs to be dynamic. The interface of the data structure must support insertion of simplices – i.e. vertices, edges and triangles – query for all triangles intersecting an edge, nearest neighbour queries for a vertex to a given vertex and enumeration of all simplices in a given box.

We chose to use a regular grid for the spatial search, because it optimally performs in static and dynamic environments, as shown by Zachmann [109], and is easy to implement. In each grid cell we store a list of the simplices, which are partially or completely contained in the cell. In the beginning of the simplification process, we use a grid with uniform edge length of twice the average edge length of the simplicial complex, such that each simplex is in average contained in one or two cells allowing for fast insertion and removal. During the simplification process we keep track of the increasing average edge length. When the latter exceeds the grid edge length, we create a new grid with double grid edge length and re-insert all remaining simplices. For the used models, we observed that in average each simplex had to be entered in a constant number of grid cells only.

7.1.1 Simplex insertion and removal

To insert a vertex we simply compute the enclosing cell and add it to its list of contained simplices. Edges and triangles can penetrate more than one cell into which they were entered. In case of an edge, a simple incremental algorithm could be used to trace the penetrated cells along the edge. For triangles,

we implemented a not optimal but simple solution. First, we collect all cells intersected by the bounding box of the triangle. Then, we sort out the actually penetrated cells by a marching-cubes-like strategy. Each of the grid vertices from intersected cells is classified to be *above*, *below* or *outside* of the triangle in consideration. Finally, the triangle is inserted in all grid cells with at least one vertex classified as *above* and one, classified as *below*. As each triangle is in average inserted into a small number of cells the presented strategy works reasonably fast[1].

For fast removal, we store for each vertex a pointer to the list item in the enclosing cell such that it could be removed in constant time. Similarly, we keep for each edge and each triangle a list of pointers to list items such that any simplex could be removed in time proportional in the number of penetrated cells. To move simplices affected by the contraction operations, we just remove them from the grid and re-insert them.

7.1.2 Spatial queries

In order to find all triangles intersecting an edge, we determine all cells, which partially contain the edge, and enumerate all triangles, which are partially contained in these cells. To sort out the non-intersecting triangles we actually perform the intersection tests between the edge and the selected triangles.

The closest non-incident vertex to a given vertex v can be found in the grid by a region-growing strategy, starting with the cell that contains the vertex. While the so far closest vertex is further away from v than the closest not considered cell, we also consider the vertices in the closest cell.

Finally, to enumerate all simplices in a given box, we simply determine all cells intersecting the box and enumerate all partially contained simplices.

7.2 Initial clean-up of self-intersections

In this section we describe how to eliminate self-intersections in the input mesh by generation of new vertices, edges and triangles. The crucial self-intersections are the ones between edges and triangles. The vertex-edge, vertex-triangle and edge-edge intersections can be eliminated in one step and all triangle-triangle intersections coincide with two edge-triangle intersections. Our first solution tried to remove the edge-triangle intersections iteratively but did not terminate in all cases.

Therefore, we came up with a method, that minimizes the number of newly generated simplices. The basic idea is to determine on each triangle the collection of points and segments resulting from intersections with other simplices

[1] A slightly more efficient method was described by Zelinka and Graland [111].

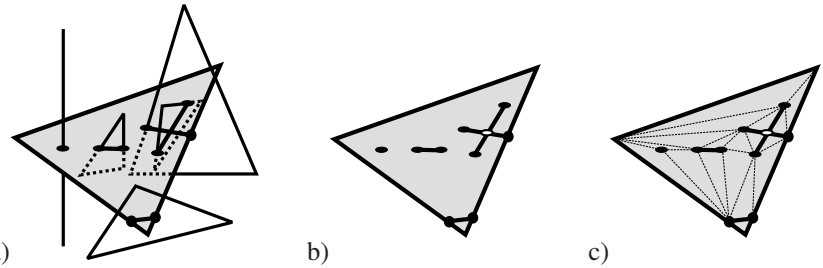


Figure 7.3: a) Intersections of triangle with other simplices. b) Creating of further vertices on intersecting segments. c) Triangulation of planar straight line graph.

as depicted in figure 7.3 a. The resulting intersection graph is triangulated in a way to include the intersection points and segments.

Suppose we want to split triangle t. The intersection with another triangle is a line segment and the intersection with an edge or a vertex is just a point. The resulting intersection graph can still have crossing edge, which are removed by insertion of additional nodes at the segment intersections as shown in figure 7.3 b.

Finally, the resulting planar straight line graph is triangulated with a standard sweep line algorithm as, for example, described by de Berg et al. [21]. Nodes that are closer than a user defined ε are snapped together in order to avoid numerical problems with very short triangle edges. Care has to be taken in order to match identical newly introduced vertices on different triangles.

7.3 Detection and prevention of self-intersections

This section describes the detection of self-intersections caused during the simplification process.

In a vertex contraction operation the two contraction vertices are contracted onto one target vertex. All simplices incident to one of the contraction vertices are called *affected* as they are sheared during the contraction. Simplices incident to both contraction vertices are called *contracted* as they are eliminated.

The affected simplices can cause two problems as illustrated in figures 7.4. Either an intersection can be caused, as depicted in figure 7.4 a, or a simplex can switch from the inside to the outside of a closed surface, as shown in figure 7.4 b. The inside-outside switch does not cause an intersection but can change the look of the model, when the switched simplex has a different colour as the enclosing surface.

Both problems can be detected by parameterizing the contraction operation

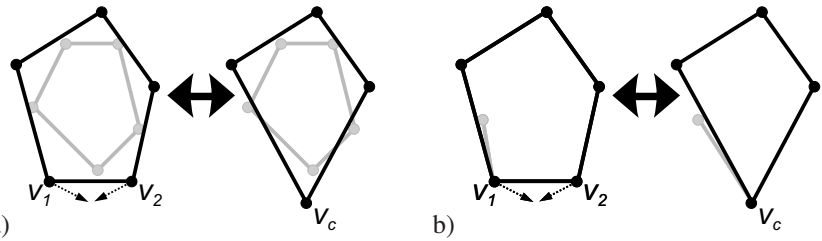


Figure 7.4: a) Intersection caused by vertex contraction. b) Inside-outside switch of differently coloured edge.

over time, as Hoppe did to generate *geomorphs* [44], and by looking for a collision between the simplices. An intersection is a collision, whereas an inside-outside switch always cause a collision. Prevention of collisions will also ensure that no intersection arises during a geomorph.

7.3.1 Classification of collisions

Suppose the contraction vertices of the current contraction operation are v_1 and v_2 and the target vertex, on which they are contracted, is v_c. Let Σ_1 be the set of affected simplices incident only to v_1 and Σ_2 the set incident to v_2. Σ_{12} is the set of contracted simplices incident to both v_1 and v_2. The remaining simplices are called *stationary* and collected in the set Σ_0. Only the stationary simplices in a bounding box containing affected and contracted simplices and the target vertex can cause a collision. These are collected via a range query from the spatial data structure.

We parameterize the vertex contraction operation over the time interval $t \in [0,1]$ by specifying the movement of the contraction vertices

$$v_j(t) = v_j + t \cdot (v_c - v_j), j \in \{1,2\}, \tag{7.1}$$

where we distinguish between the starting locations v_j of the contraction vertices and their time evolution $v_j(t)$ only by the additional time parameter. As we cleaned up all intersections in the beginning, we can start off with the precondition that there is no collision at $t = 0$. We distinguish five types of collisions:

- *hit collisions of the first kind* between a simplex from $\Sigma_1 \cup \Sigma_2$ and a simplex from Σ_0,
- *hit collisions of the second kind* between a simplex from Σ_{12} and a simplex from Σ_0,
- *fan collisions* between two simplices from Σ_1 or two simplices from Σ_2,

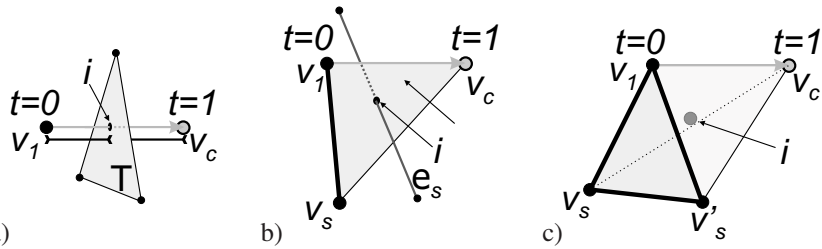


Figure 7.5: Detection of hit collisions of the first kind with the time sweep of the affected vertex (a), edge (b) or triangle (c) and calculation of collision time.

- *contraction collisions of the first kind* between a simplex from Σ_1 and a simplex from Σ_2 and
- *contraction collisions of the second kind* between a simplex from $\Sigma_1 \cup \Sigma_2$ and a simplex from Σ_{12} or between two simplices from Σ_{12}.

7.3.2 Efficient computation of collisions

To reduce the number of to be checked collisions and the number of to be discussed collision types, we state the following lemma, which directly follows from the continuous parametrization of the pair contraction operation.

Lemma 7.1
If a pair contraction is applied to a simplicial complex without self-intersections, any collision between an edge and a triangle or between two triangles is preceded by or coincides with a collision between a vertex and a simplex or between two edges.

Lemma 7.1 implies that it is sufficient to test for collisions of the *first class*, i.e. between a vertex and another simplex or between two edges, because if any other collision arises there will also be one of the first class happening even earlier in time.

If we split the pair contraction into two phases by, first, dragging vertex v_1 with fixed v_2 onto v_c and by, second, dragging v_2 onto v_c, only hit collisions of the first kind can arise. All the first class hit collisions of the first kind can be detected by an intersection test between the time sweep of the affected simplex with the stationary simplex as illustrated in figures 7.5.

The time sweep of the affected vertex is a line segment, which needs to be tested against all stationary vertices, edges and triangles. Figure 7.5 a illustrates the case of a stationary triangle. Segment-vertex, segment-segment and segment-triangle intersection tests can be implemented efficiently.

In figure 7.5 b the time sweep of an affected edge is shown to be a triangle, which needs to be tested for intersection with all stationary vertices and edges. Finally, the time sweep of an affected triangle is a tetrahedron as depicted in figure 7.5 c. Here only intersection tests with stationary vertices are necessary, which reduce to a simple point inclusion test.

If we do not follow the two-phase strategy, one can additionally show that fan collisions and contraction collisions of the second kind always coincide with another type of collision and, therefore, do not have to be tested. The intersection tests for hit collisions of the second kind and contraction collisions of the first kind are slightly more complicated and are skipped here.

7.4 Avoidance of self-intersections

The simple prevention strategy of just discarding operations that cause a collision does not allow the generation of very coarse approximations with low error. As shown in section 7.5, the approximation error grows much faster if a lot of low error operations have to be discarded, and furthermore is it possible that the simplification process reaches a point when all possible operation would cause a self-intersection. For models like the dressed man most operations of vertices close to the cloths won't be valid in a late stage of the simplification process. Thus only the head, hands and feet can be reduced further yielding a bad approximation of the head and hand, as can be seen in figures 7.6.

To actually avoid the large number of invalid operations, it is either possible to look for a different target position that does not cause a self-intersection or to move the part of the mesh, into which the contracted part bumped, or do both. We decided to only change the target position, as in this case the progressive representation of the model can be stored in exactly the same way as if no self-intersection avoidance had been performed.

Thus we simply try to find a target location that does not cause a self-intersection. In some situations this is not possible at all. But in most situations there is a so called *valid region* of valid target positions onto which the two contraction vertices can be contracted without causing a self-intersection. The target position, which is optimal with respect to quadric error metric, causes a self-intersection and is, therefore, not in the valid region. What we are looking for is the location in the valid region that minimizes the quadric error metric. As the explicit computation of the valid region is very complicated, we came up with three approximate solutions:

- *First hit:* For all collision tests introduced in the previous section one automatically computes the parameter value $t = t_c$, when the collision arises. By minimizing the time parameter over all colliding affected simplices one can determine the time parameter $t_{\min}$ when the first collision

arises. In the first hit strategy we guess the new collision-free target location from the location of the contraction vertex incident to the simplex causing the collision at time $t_{\min}$. In the case this target location is again causing a collision, we use the same strategy again up to five times before we give up.

- *Barycentric sampling:* Here we sample possible target locations only on the triangle spanned by the two contraction vertices and the QEM-optimal target location. We sample the triangle on fourteen further locations, that result from two one to four subdivisions. We sort the sample locations by increasing QEM error and perform collision checks until the first yields a valid target location or until all of them failed and the operation has to be discarded.

- *Extensive sampling:* As shown in section 7.5, it is not obvious at all how to do an extensive sampling of the possible target location in order to keep the approximation error low. In our extensive sampling strategy we sample the space around the optimal target location in 26 directions given by the vertices, edge centers and face centers of an axis aligned octahedron, i.e. all possible directions with one, two or three ± 1 as coordinates. We sample an iso-surface of the quadric error metric, where it is hit by the 26 directions. Let $\varepsilon_{1/2}^2$ be the quadric error values at the initial locations of the contraction vertices, and ε_c^2, the one at the optimal target location. Let

$$\varepsilon_\alpha^2 = \alpha \cdot min(\varepsilon_1^2, \varepsilon_2^2) + (1-\alpha) \cdot \varepsilon_c^2, \tag{7.2}$$

  We determine the minimum value for α by an interval subdivision of the interval $[0,1]$, where we check for each α, if at least one valid target location exists.

In case a valid target location is found by one of the strategies, we evaluate the quadric error metric at the new approximation of the target position and re-insert the operation into the priority queue. Figure 7.6 c shows the model of a man simplified with avoidance of self-intersections.

7.5 Results

We tested our intersection-free simplification algorithm on four test data sets: a dressed man, a dressed woman[2], two nested spheres and the bunny – as an example of a model that does not cause self-intersections. Figure 7.2 c shows the reduced nested sphere model. In figures 7.6 and 7.7, the man and woman

[2]The models of a man and a woman were kindly provided by the *Virtual Try-On* project.

Models	Man	Woman	Spheres	Bunny
Original number of vertices	17 060	19 498	1 028	34 838
Reduced number of vertices	600	370	100	500
Time (s) – no detection	1.87	2.02	1.23	1.29
Time (s) – prevention	7.52	8.45	4.31	3.12
Time (s) – first hit	10.18	10.71	4.43	3.18
Time (s) – barycentric	18.52	17.67	7.06	3.06
Time (s) – extensive	63.77	72.89	14.02	3.62
Hausdorff dist. – no detection	0.1050	0.1597	0.4841	0.1069
Hausdorff dist. – prevention	0.1293	0.2027	0.4719	0.1070
Hausdorff dist. – first hit	0.1246	0.1850	0.4894	0.1074
Hausdorff dist. – barycentric	0.1183	0.1648	0.4816	0.1072
Hausdorff dist. – extensive	0.1195	0.1708	0.4857	0.1071

Table 7.1: Experimental results. Reduction times are per 1000 pair contractions and where measured on an Athlon processor with 1.2 GHz.

models are depicted. The bunny did not produce any self-intersections and is not shown.

Table 7.1 gathers the experimental results. The first two rows show the number of vertices of the original and simplified models. The next five rows show the simplification time without collision test, with collision prevention and with the three different avoidance strategies. The last four rows show the Hausdorff distances between the simplified and original model achieved either without collision test, with collision prevention or with the three different avoidance strategies, measured using the *Mesh* tool [3].

For the bunny model only two collisions arose. The increase of running time is due only to the insertion and removal of the simplices to/from the spatial data structure. For the other models there are a lot of collision tests because of the large portion of double layered parts. The measurements of the Hausdorff distances show how the problem of the invalidation of operations by the simple collision detection strategy can be nearly completely avoided by the barycentric sampling.

7.6 Related work

As most of the models used with current simplification algorithms do not contain close layers of different colour, few works considered the problem of self-intersections at all. Typically, only the normal-flip test proposed by Ronfard and Rossignac [82] is performed, which checks if the normals of the incident

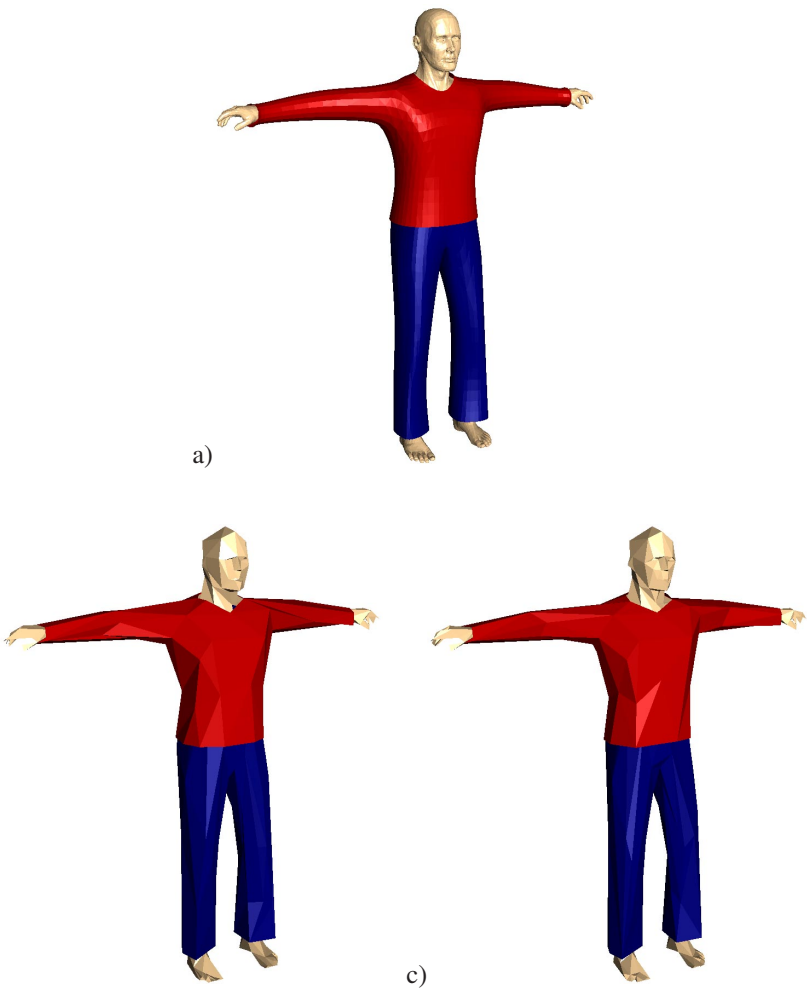


Figure 7.6: Dressed man model: original containing 17 060 vertices (a), simplified to 600 vertices with prevention (b) and avoidance (c) of self-intersections.

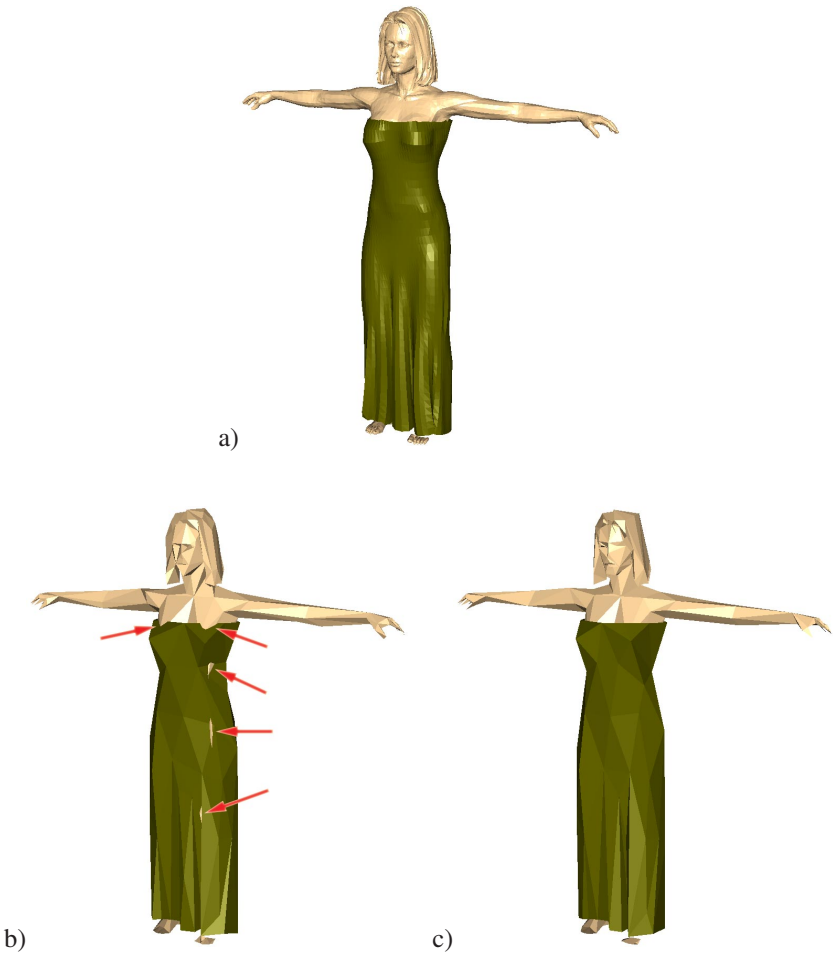


Figure 7.7: Dressed woman model: original containing 19 498 vertices (a), simplified to 370 vertices without collision test (b) and with collision avoidance (c).

triangles change their direction more than by a maximum angular deviation and, in this way, allows to prevent local self-intersections. However, this does not allow to produce acceptable simplifications of the very important class of meshes that represent dressed people.

A method that can at least prevent global self-intersections was described by Cohen at al. [20]. In order to ensure a one-sided Hausdorff distance of ε from the simplified model to the original model, two offset surfaces of $\pm\varepsilon$ around the original model are defined. To prevent self-intersections of the offset surfaces a different ε_i is specified for each vertex and is reduced from the user specified epsilon, until no self-intersections of the so called *simplification envelopes* remain. During simplification, the prevention strategy, as described in the introduction, is performed to prevent, on the one hand, self-intersections and, on the other hand, an increase in the one-sided Hausdorff distance over ε by enforcing all triangles to stay within the tolerance volume enclosed by the envelopes. Simplification envelopes are not efficient in a setting where ε changes.

Zelinka and Garland [111] discretize the tolerance volume around the original mesh on a regular grid to accelerate the validity tests for the atomic simplification operations. An increase in ε is achieved by growing the discretized tolerance volume by one cell in all directions. They do not check self-intersections of the tolerance volume and, therefore, do not prevent global selfintersections.

Guéziec [35] dynamically keeps a tolerance volume around the simplified mesh such that the one-sided Hausdorff distance from the original model to the simplified mesh can be controlled. In this approach, global self-intersections are neither avoided.

Both the approaches of Guéziec and Zelinka/Garland could probably be generalized to avoid self-intersections. But all tolerance volume based approaches to the prevention of self-intersections have one disadvantage. The space between two close surface layers has to be split into two parts, one for each layer. This split is done in the initialization stage without knowing potential self-intersection problems. The optimal target location of an edge collapse in one layer could then fall out of the corresponding tolerance volume, although no self-intersection is caused. Thus, the tolerance-volume-based approaches can discard the operation with the smallest approximation error even if it does not cause a self-intersection. Our approach on the other hand does not report any non-existent self-intersections.

The simplification algorithm creating progressive tetrahedralizations described by Staadt et. al [91] avoids self-intersections of the boundary surface of the tetrahedral mesh. They argue that in the case of volume boundary meshes these self-intersections can only arise at sharp boundary edges and restrict their intersection tests to these edges. Their assumption does actually not even hold for volume boundary meshes as a simple example shows: take a solid cube and

cut out a very fine spherical layer. During the simplification the two sides of the layer can easily intersect without the vicinity of any sharp edges. In this case a method like ours is necessary.

7.7 Summary

We presented the first approach to globally prevent and avoid self-intersections during mesh simplification. We based our implementation on the standard vertex contraction approach and support non-manifold meshes and modification of the topology. Other simplification approaches could be supported as well. Progressive transmission and several level-of-detail algorithms can be supported self-intersection-free without any modifications, as we do not introduce any new operations but only change the target locations of the pair contractions.

From the introduced strategies to find valid target locations in the case of self-intersections, the first-hit strategy proved to be very efficient in terms of running time and achieved approximation quality.

The proposed method is the first that can create high-quality simplified versions of dressed people.

Part IV

Conclusion and future work

Chapter 8

Conclusion

The topic of this thesis are algorithms for mesh simplification and mesh repair. Since these areas are quite popular in Computer Graphics, and, therefore, a lot of research was already done in both fields, in our work we addressed only several particular problems of mesh simplification and mesh repair. Presented here is a collection of techniques that allow to produce high-quality models with properties, which are important prerequisites for most of the rendering and processing methods.

8.1 Mesh repair

In chapter 3, we introduced a mesh repair method that consistently orients all face normals of an arbitrary polygonal mesh and simultaneously ensures that as much as possible polygons of the model are seen with their front faces from most viewpoints. In our algorithm, we combine the traditional proximity-based approach with our new visibility-based approach. This allows us to produce the desirable solution for most practical cases of polygonal meshes containing inconsistently oriented normals, except the models with many overlapping and coplanar polygons.

Another mesh repair method, which removes the inconsistency of vertex connectivity, was presented in chapter 4. Here, we interpret the problem of generation of topologically connected polygonal models as a mesh boundary simplification task. In addition to the vertex contraction and edge collapse operations traditional for mesh simplification, we introduce a new vertex-edge contraction operation, which provides additional flexibility. We use this new operator to simplify the boundaries of the mesh, thereby we manage to remove various artefacts, such as T-vertices, degenerate triangles, narrow gaps and cracks. By applying the vertex-edge operator according to an increasing error, we successively close the gaps and holes in the model.

8.2 Mesh simplification

In chapter 5, we presented a technique that generates high-quality simplified models. It utilises the generalized pair contractions – contraction of a vertex with another vertex, an edge or a triangle and contraction of two edges, which allow to repair cracks and self-intersections and to sew unconnected components with lesser error than standard vertex contraction. Therefore, our new method is particularly useful for the simplification of the models consisting of a large number of unconnected parts. In addition to its ability to repair meshes in an intuitive and efficient way, our algorithm often results in much better decimation than previous simplification techniques.

Using the above high-quality simplification method, we implemented a simple and fast out-of-core mesh simplification algorithm, which we described in chapter 6 of this thesis, that is capable to guarantee a given geometric distance between original and simplified model. Our topological simplification is controlled based on a geometric error. Since we use generalized pair contractions and cut the model at octree cell boundaries, we do not accumulate the triangles that intersect the octree cells. Despite the fact that the computation time of our algorithm is higher compared to recent approaches, the gain in quality and/or reduction rate is significant.

Finally, in chapter 7, we introduced the first approach to globally prevent and avoid self-intersections that can occur during mesh simplification. We do it by parameterizing the contraction operations over time and by detecting collisions of affected simplices. In the case of a collision, we determine a new target position that avoid the collision using one of the three different strategies. Our method produces high-quality simplified meshes without causing any new self-intersections and is especially suitable to simplify models with close layers, such as dressed people.

Chapter 9

Future work

Since the development and publication at different conferences of the methods collected in this thesis, a lot of research has been performed both in the field of mesh simplification and in the field of mesh repair. In sections 3.3, 4.4, 5.7, 6.3 and 7.6, we provided an overview of the techniques related to the topics of this thesis. In addition to the work done previously, this also included several newer approaches, some of which contain references and/or directly relate to our techniques.

In the field of mesh repair, we have been dealing with few particular types of artefacts that occur in polygonal models. However, a growing number of models contain errors of various types. An incomplete list of such artefacts includes: cracks, holes, T-joints, overlaps, dangling walls, duplicated geometry, self-intersections, inconsistent normal orientation, invisible polygons, degenerated faces, concavities, etc. A natural direction for future work would be to develop mesh repair techniques that handle errors of other types and, furthermore, can repair many types of errors simultaneously. This, for example, try to do several newer methods based on volumetric techniques.

An aspect of mesh repair not covered in our work is the integration of user interaction. Often, is is not easy to automatically decide if a particular mesh structure is an artefact or a desired feature. In some practical cases, the correct solution can not be found without human decision, and development of techniques that provide the user an easy and intuitive possibilities to interactively make a selection is another topic of interest in the area of mesh repair.

With regards to the techniques presented in this thesis, our normal-orienting method could produce undesirable results for some models, which contain many overlapping and coplanar polygons, as mentioned in section 3.4. Solution of this issue would be a task for future work.

As of mesh simplification, a drawback of our algorithms based on generalized pair contractions is quite high memory consumption. This issue was

discussed in section 2.2.3.1, and the search of a reasonable compromise between memory use and method's productivity would surely be an advantage. However, this is only an implementation issue.

Finally, as we mentioned in section 6.4, the computation time of our out-of-core simplification method is higher compared to other recent approaches. Acceleration of this algorithm without any loss in quality and reduction rate could be another task for future research.

Bibliography

[1] 3D Systems, Inc., Valencia, CA. *Stereolitography interface specification*, October 1989. p/n 500650S01-00.

[2] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics (Proceedings of SIGGRAPH 1987)*, 21(4):55–64, July 1987.

[3] Nicolas Aspert, Diego Santa-Cruz, and Touradj Ebrahimi. Mesh: Measuring errors between surfaces using the hausdorff distance. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, volume I, pages 705–708. IEEE Computer Society Press, 2002.

[4] Sergei Azernikov and Anath Fischer. Surface reconstruction of freeform objects based on hierarchical space decomposition. In *4th Israel-Korea Bi-National Conference on Geometric Modeling and Computer Graphics – Conference Proceedings*. Tel Aviv University, February 2003.

[5] Sergei Azernikov and Anath Fischer. Efficient surface reconstruction method for distributed cad. *Computer-Aided Design*, 36(9):799–808, August 2004.

[6] Sergei Azernikov, Alex Miropolsky, and Anath Fischer. Surface reconstruction of freeform objects based on multiresolution volumetric method. In *Proceedings of the 8th ACM Symposium on Solid Modeling and Applications*. ACM Press, June 2003.

[7] Gill Barequet and Subodh Kumar. Repairing CAD models. In Roni Yagel and Hans Hagen, editors, *IEEE Visualization '97 Conference Proceedings*, pages 363–370. IEEE Computer Society Press, 1997.

[8] Bruce G. Baumgart. Winged-edge polyhedron representation. Technical Report STAN-CS-320, Stanford University, October 1972.

[9] Bruce G. Baumgart. A polyhedron representation for computer vision. In *Proceedings of the 1975 National Computer Conference*, volume 44 of *AFIPS Proceedings*, pages 589–596. AFIPS Press, May 1975.

[10] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Claudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, October 1999.

[11] Stephan Bischoff, Darko Pavić, and Leif Kobbelt. Automatic restoration of polygon models. *ACM Transactions on Graphics*, 24(4):1332–1352, October 2005.

[12] Mario Botsch, Mark Pauly, Christian Rössl, Stephan Bischoff, and Leif Kobbelt. Geometric modeling based on triangle meshes. In *ACM SIGGRAPH 2006 Courses*. ACM Press, August 2006.

[13] Geoffrey Butlin and Clive Stops. CAD data repair. In *5th International Meshing Roundtable*, pages 7–12, 1996.

[14] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed edges – a scalable representation for triangle meshes. *Journal of Graphics Tools*, 3(4):1–12, 1998.

[15] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Enhancing digital documents by including 3D-models. *Computers & Graphics*, 22(6):655–666, 1998.

[16] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6(5):485–524, November 1991.

[17] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[18] Paolo Cignoni, Claudio Rocchini, Claudio Montani, and Roberto Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2002.

[19] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[20] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Kumar Agarwal, Jr. Frederick P. Brooks, and William V.

Wright. Simplification envelopes. In *SIGGRAPH 1996 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 119–128. ACM Press/Addison-Wesley, August 1996.

[21] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, June 1997.

[22] Hélène Desaulniers and Neil F. Stewart. An extension of manifold boundary representation to the r-Sets. *ACM Transaction on Graphics*, 11(1):40–60, January 1992.

[23] Charles M. Eastman. Introduction to computer-aided design, 1982. Course notes, Carnegie-Mellon University.

[24] Charles M. Eastman and Kevin J. Weiler. Geometric modeling using the Euler operators. In *Proceedings of the 1st Annual Conference on Computer Graphics in CAD/CAM Systems*, pages 248–259. MIT Press, May 1979.

[25] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification and rendering. *Computer Graphics Forum*, 19(3):139–150, September 2000.

[26] Carl Erikson and Dinesh Manocha. HLODs for faster display of large static and dynamic environments. In *ACM Symposium on Interactive 3D Graphics*, 2000.

[27] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, second edition, August 1995.

[28] Steven Fortune and Christopher J. van Wyk. Efficient exact arithmetic for computational geometry. In *Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pages 163–172, May 1993.

[29] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH 1997 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 209–216. ACM Press/Addison-Wesley, August 1997.

[30] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98 Conference Proceedings*, pages 263–270. IEEE Computer Society Press, 1998.

[31] Michael Garland and Eric Shaffer. A multiphase approach to efficient surface simplification. In *IEEE Visualization (Proceedings of the conference on Visualization'02)*, pages 117–124. IEEE Computer Society Press, 2002.

[32] Paul-Louis George. *Automatic Mesh Generation: Applications to Finite Element Methods*. John Wiley & Sons, September 1991.

[33] Andrew S. Glassner. Maintaining winged-edge models. In James Arvo, editor, *Graphics Gems II*, pages 191–201. Academic Press, October 1991.

[34] Stefka Gueorguieva and David Marcheix. Non-manifold boundary representation for solid modelling. In *Proceedings of the 1994 International Computer Symposium*, December 1994.

[35] André Guéziec, Gabriel Taubin, Francis Lazarus, and Bill Horn. Cutting and stitching: Converting sets of polygons to manifold surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 7(2):136–151, 2001.

[36] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transaction on Graphics*, 4(2):74–123, April 1985.

[37] Igor Guskov and Zoe J. Wood. Topological noise removal. In *Graphics Interface*, 2001.

[38] Michael Guthe, Pavel Borodin, and Reinhard Klein. Efficient view-dependent out-of-core visualization. In Jizhou Sun and Zhigeng Pan, editors, *Fourth International Conference on Virtual Reality and Its Application in Industry*, volume 5444 of *Proceedings of SPIE*, pages 428–438. SPIE — The International Society for Optical Engineering, 2004.

[39] Michael Guthe, Jan Meseth, and Reinhard Klein. Fast and memory efficient view-dependent trimmed nurbs rendering. In S. Coquillart, H.-Y. Shum, and S.-M. Hu, editors, *Pacific Graphics 2002*, pages 204–213. IEEE Computer Society Press, October 2002.

[40] Michael M. Heck. *VRML 2.0 for Open Inventor Programmers*. Template Graphics Software, July 1997.

[41] Martin Held. Efficient and reliable triangulation of polygons. In Franz-Erich Wolter and Nicholas M. Patrikalakis, editors, *Proceedings of the Computer Graphics International Conference 1998*, pages 633–643. IEEE Computer Society Press, June 1998.

[42] Franck Hétroy, Stéphanie Rey, Carlos Andújar, Pere Brunet, and Àlvar Vinacua. Mesh repair with topology control. Technical Report 6535, INRIA – Institut National de Recherche en Informatique et en Automatique, May 2008.

[43] Christoph M. Hoffmann. *Geometric and Solid Modeling: An Introduction*. Morgan Kaufmann Publishers, July 1989.

[44] Hugues Hoppe. Progressive meshes. In *SIGGRAPH 1996 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 99–108. ACM Press/Addison-Wesley, August 1996.

[45] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH 1997 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 189–198. ACM Press/Addison-Wesley, August 1997.

[46] Hugues Hoppe. Smooth view-dependant level-of-detail control and its application to terrain rendering. In *IEEE Visualization '98 Conference Proceedings*, pages 35–52. IEEE Computer Society Press, October 1998.

[47] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics*, 26(2):71–78, 1992.

[48] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In James T. Kajiya, editor, *SIGGRAPH 1993 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 19–26. ACM Press/Addison-Wesley, August 1993.

[49] Martin Isenburg, Stefan Gumhold, and Jack Snoeyink. Processing sequences: a new paradigm for out-of-core processing on large meshes, 2003. Preprint available at `http://www.cs.unc.edu/~isenburg/papers/igs-ps-03.pdf`.

[50] Martin Isenburg, Peter Lindstrom, Stefan Gumhold, and Jack Snoeyink. Large mesh simplification using processing sequences. In Greg Turk, Jarke J. van Wijk, and Robert J. Moorhead II, editors, *IEEE Visualization (Proceedings of the 14th IEEE Visualization 2003)*, pages 465–472. IEEE Computer Society Press, October 2003.

[51] Tao Ju. Robust repair of polygonal models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2004)*, 23(3):888–895, August 2004.

[52] Brian W. Kernighan and Christopher J. van Wyk. Extracting geometrical information from architectural drawings. In *Proceedings of the Workshop on Applied Computational Geometry*, pages 82–87, May 1996.

[53] Lutz Kettner. Designing a data structure for polyhedral surfaces. In *Proceedings of the 14th Annual Symposium on Computational Geometry*, pages 146–154. ACM Press, June 1998.

[54] David G. Kirkpatrick, Maria M. Klawe, and Robert E. Tarjan. Polygon triangulation in $o(nloglogn)$ time with simple data structures. In *Proceedings of the 6th Annual Symposium on Computational Geometry*, pages 34–43. ACM Press, June 1990.

[55] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In Roni Yagel and Gregory M. Nielson., editors, *IEEE Visualization '96 Conference Proceedings*, pages 311–318. IEEE Computer Society Press, 1996.

[56] Reinhard Klein, Andreas Schilling, and Wolfgang Straßer. Illumination dependent refinement of multiresolution meshes. In *Proceedings of Computer Graphics International (CGI'98)*, pages 680–687. IEEE Computer Society Press, 1998.

[57] Robert Laurini and Fran coise Milleret-Raffort. Topological reorganization of inconsistent geographical databases: a step towards their certification. *Computer and Graphics*, 18(6):803–813, 1994.

[58] Pascal Lienhardt. Topological models for boundary representation: A comparison with *n*-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–82, February 1991.

[59] Peter Lindstrom. Out-of-core simplification of large polygonal models. In *SIGGRAPH 2000 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 259–262. ACM Press/Addison-Wesley, July 2000.

[60] Peter Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 93–102, 239. ACM Press, 2002.

[61] Peter Lindstrom and Claudio T. Silva. A memory insensitive technique for large model simplification. In *IEEE Visualization 2001 Conference Proceedings*, pages 121–126, 550. IEEE Computer Society Press, October 2001.

[62] Kok-Lim Low and Tiow Seng Tan. Model simplification using vertex-clustering. In *Symposium on Interactive 3D Graphics*, pages 75–82, 188. ACM Press, 1997.

[63] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, July 2001.

[64] David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, May 2001.

[65] Martti Mäntylä. A note on the modeling space of Euler operators. *Computer Vision, Graphics, and Image Processing*, 26(1):45–60, April 1984.

[66] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, June 1988.

[67] Martti Mäntylä and Reijo Sulonen. GWB: A solid modeler with Euler operators. *IEEE Computer Graphics and Applications*, 2(7):17–31, September 1982.

[68] Sara McMains, Joseph M. Hellerstein, and Carlo H. Séquin. Out-of-core build of a topological data structure from polygon soup. In David C. Anderson, editor, *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications*. ACM Press, June 2001.

[69] David E. Muller and Franco P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.

[70] T. M. Murali and Thomas A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *Symposium on Interactive 3D Graphics*, pages 155–162, 196, 1997.

[71] James D. Murray and William vanRyper. *Encyclopedia of Graphics File Formats*. O'Reilly and Associates, 2nd edition, May 1996.

[72] Atul Narkhede and Dinesh Manocha. Fast polygon triangulation based on Seidel's algorithm. In Alan W. Paeth, editor, *Graphics Gems V*, pages 394–397. Academic Press/Morgan Kaufmann, May 1995.

[73] Xiujun Ni. *Free-Form Solid Modeling Using a Hybrid CSG/BRep Environment*. PhD thesis, University of Leeds, December 1990.

[74] Xiujun Ni and M. Susan Bloor. Performance evaluation of boundary data structures. *IEEE Computer Graphics and Applications*, 14(6):66–77, November 1994.

[75] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. Technical Report GITGVU-99-37, Georgia Institute of Technology, 1999.

[76] Fakir S. Nooruddin and Greg Turk. Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):191–205, April 2003.

[77] Marcin Novotni and Reinhard Klein. Computing geodesic distances on triangular meshes. *Journal of WSCG*, 10(2):341–347, February 2002.

[78] William Palm. Rapid prototyping primer. `http://www.me.psu.edu/lamancusa/rapidpro/primer/chapter2.htm`, July 2002. In *The Learning Factory* (`http://www.lf.psu.edu`). Pennsylvania State University.

[79] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. In *SIGGRAPH 1997 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 217–224. ACM Press/Addison-Wesley, August 1997.

[80] Chris Prince. Progressive meshes for large models of arbitrary topology. master's thesis, department of computer science and engeneering, university of washington, seattle, 2000.

[81] Aristides A. G. Requicha and Herbert B. Voelcker. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of the IEEE*, 73(1):30–44, January 1985.

[82] Rémi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum (Proceedings of Eurographics '96)*, 15(3):67–76, September 1996.

[83] Linda Rose and Diane Ramey. *Wavefront file formats, version 4.0 RG-10-004*. Wavefront Technologies Inc., Santa Barbara, CA, first edition, 1993.

[84] Jarek Rossignac and Paul Borrel. Multi-resolution 3D approximations for rendering. In Bianca Falcidieno and Tosiyasu L. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 455–465. Springer Verlag, June 1993.

[85] Gernot Schaufler and Wolfgang Stürzlinger. Generating multiple levels of detail from polygonal geometry models. In Martin Göbel, editor, *Virtual Environments '95 (Proceedings of 2nd Eurographics Workshop on Virtual Environments)*, pages 33–41. Springer Verlag, January 1995.

[86] Mark Segal. Using tolerances to guarantee valid polyhedral modeling results. In Forest Baskett, editor, *SIGGRAPH 1990 Conference Proceedings: Computer Graphics Annual Conference Series*, pages 105–114. ACM Press/Addison-Wesley, August 1990.

[87] Raimund Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, July 1991.

[88] Xuejun Sheng and Ingo R. Meier. Generating topological structures for surface models. *IEEE Computer Graphics and Applications*, 15(6):35–41, 1997.

[89] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997.

[90] Vinod Srinivasan, Ergun Akleman, and John Keyser. Topological construction of 2-manifold meshes from arbitrary polygonal data. Technical report, Visualization Sciences Program, College of Architecture, Texas A&M University, January 2004.

[91] Oliver G. Staadt and Markus H. Gross. Progressive tetrahedralizations. In *IEEE Visualization '98 Conference Proceedings*, pages 397–404. IEEE Computer Society Press, October 1998.

[92] Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 239–246, 1993.

[93] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics (Proceedings of SIGGRAPH 1987)*, 21(4):153–162, July 1987.

[94] Greg Turk. *The PLY Polygon File Format*. Georgia Institute of Technology, 1998.

[95] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. *Computer Graphics*, 28(Annual Conference Series):311–318, 1994.

[96] Daniel Veleba and Petr Felkel. Survey of errors in surface representation and their detection and correction. In Václav Skala, editor, *WSCG'2007 Short Communications Proceedings*, pages 71–78. UNION Agency – Science Press, January 2007.

[97] Karsten Weihe and Thomas Willhalm. Why CAD data repair requires discrete algorithmic techniques. In Kurt Mehlhorn, editor, *Proceedings of 2nd International Workshop on Algorithm Engineering*, pages 1–12. Max-Planck-Institut für Informatik, August 1998.

[98] Kevin J. Weiler. Adjacency relationships in boundary graph-based solid models. Unpublished, June 1983.

[99] Kevin J. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, January 1985.

[100] Kevin J. Weiler. Non-manifold geometric boundary modeling. SIGGRAPH 1987 Tutorial on Advanced Solid Modeling, July 1987.

[101] Kevin J. Weiler. Boundary graph operators for non-manifold geometric modeling topology representations. In Michael J. Wozny, Harry W. McLaughlin, and José L. Encarnação, editors, *Geometric Modeling for CAD Applications*, pages 37–66. North-Holland, May 1988.

[102] Kevin J. Weiler. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. In Michael J. Wozny, Harry W. McLaughlin, and José L. Encarnação, editors, *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, May 1988.

[103] Josie Wernecke. *Open Inventor Nodes Quick Reference, Release 2.0*. Silicon Graphics, July 1994.

[104] Nathaniel Williams, David Luebke, Jonathan D. Cohen, Michael Kelley, and Brenden Schubert. Perceptually guided simplification of lit, textured meshes. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 113–121. ACM Press, 2003.

[105] Paul R. Wilson. Euler formulas and geometric modeling. *IEEE Computer Graphics and Applications*, 5(8):24–36, August 1985.

[106] Tony C. Woo. A combinatorial analysis of boundary data structure schemata. *IEEE Computer Graphics and Applications*, 5(3):19–27, March 1985.

[107] Jianhua Wu and Leif Kobbelt. A stream algorithm for the decimation of massive meshes. In Torsten Möller and Colin Ware, editors, *Graphics Interface 2003 Proceedings*, pages 185–192. A K Peters Ltd., June 2003.

[108] Shin-Ting Wu. Considerations about a minimal set of non-manifold operators. In *Proceedings of the 1990 IFIP/RPI Geometric Modeling Conference*, 1990.

[109] Gabriel Zachmann. Optimizing the collision detection pipeline. In *The First International Game Technology Conference (GTEC'01)*, January 2001.

[110] Gabriel Zachmann and Elmar Langetepe. Geometric data structures for computer graphics. In *ACM SIGGRAPH 2003 Courses*. ACM Press, July 2003. Tutorial.

[111] Steve Zelinka and Michael Garland. Permission grids: practical, error-bounded simplification. *ACM Transactions on Graphics*, 21(2):1–25, April 2002.

[112] Kaichi Zhou, Eugene Zhang, Jiří Bittner, and Peter Wonka. Visibility-driven mesh analysis and visualization through graph cuts. *IEEE Transactions on Visualization and Computer Graphics (IEEE Visualization 2008 Conference Proceedings)*, 14(6):1667–1674, November 2008.

Author's publications

[1] Pavel Borodin, Mikhail Galanin, and Igor Dubovitski. The numerical solution of the problem of impulse thermal pressure on the layered elastic medium in spherically symmetric and two-dimensional cases. Technical Report 41, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 1997. In Russian.

[2] Pavel Borodin and Mikhail Galanin. The nonstationary coupled problem of thermoelasticity in different spatial approximations. *Mathematical Modeling, Moscow*, 10(3):61–82, 1998. In Russian.

[3] Pavel Borodin and Reinhard Klein. Progressive meshes with controlled topology modifications. In *Proceedings of the 1st OpenSG Symposium*, January 2002. http://www.opensg.org/OpenSGPLUS/symposium/Papers2002/.

[4] Pavel Borodin, Marcin Novotni, and Reinhard Klein. Progressive gap closing for mesh repairing. In John Vince and Rae Earnshaw, editors, *Advances in Modelling, Animation and Rendering (Proceedings of the Computer Graphics International Conference 2002)*, pages 201–213. Springer-Verlag, July 2002.

[5] Stefan Gumhold, Pavel Borodin, and Reinhard Klein. Intersection free simplification. In *4th Israel-Korea Bi-National Conference on Geometric Modeling and Computer Graphics — Conference Proceedings*, pages 11–16. Tel Aviv University, February 2003.

[6] Pavel Borodin, Stefan Gumhold, Michael Guthe, and Reinhard Klein. High-quality simplification with generalized pair contractions. In *GraphiCon 2003 — Conference Proceedings*, pages 147–154. Moscow State University, September 2003.

[7] Pavel Borodin, Michael Guthe, and Reinhard Klein. Out-of-core simplification with guaranteed error tolerance. In Thomas Ertl, Bernd Girod,

Günther Greiner, Heinrich Niemann, Hans-Peter Seidel, Eckehard Steinbach, and Rüdiger Westermann, editors, *Vision, Modeling and Visualization 2003 — Proceedings*, pages 309–316. Akademische Verlagsgesellschaft Aka GmbH, Berlin, November 2003.

[8] Stefan Gumhold, Pavel Borodin, and Reinhard Klein. Intersection free simplification. *International Journal of Shape Modeling (IJSM)*, 9(2):155–176, December 2003.

[9] Michael Guthe, Pavel Borodin, and Reinhard Klein. Efficient view-dependent out-of-core visualization. In Jizhou Sun and Zhigeng Pan, editors, *Fourth International Conference on Virtual Reality and Its Application in Industry*, volume 5444 of *Proceedings of SPIE*, pages 428–438. SPIE — The International Society for Optical Engineering, 2004.

[10] Pavel Borodin, Gabriel Zachmann, and Reinhard Klein. Consistent normal orientation for polygonal meshes. In *Proceedings of the Computer Graphics International Conference 2004*, pages 18–25. IEEE Computer Society, June 2004.

[11] Michael Guthe, Pavel Borodin, Ákos Balázs, and Reinhard Klein. Real-time appearance preserving out-of-core rendering with shadows. In Alexander Keller and Henrik W. Jensen, editors, *Rendering Techniques 2004 (Proceedings of the Eurographics Symposium on Rendering 2004)*, pages 69–79, 409. Eurographics Association, June 2004.

[12] Michael Guthe, Pavel Borodin, and Reinhard Klein. Fast and accurate Hausdorff distance calculation between meshes. *Journal of WSCG*, 13(2):41–48, February 2005.

[13] Pavel Borodin, Alexander Greß, and Reinhard Klein. Visualization aspects in the MERCW project. In *Proceedings of the 2nd US/EU-Baltic International Symposium*, May 2006.

[14] Natalia Goncharova, Pavel Borodin, and Alexander Greß. GIS for planning, navigation acquisition and visualization of results for the study of chemical munition dumpsites in the Baltic Sea. In Yangbo Chen, Ian Cluckie, and Kaoru Takara, editors, *Proceedings of the 2nd International Conference of GIS/RS in Hydrology, Water Resources and Environment (ICGRHWE '07) / 2nd International Symposium on Flood Forecasting and Management with GIS and Remote Sensing (FM2S '07)*, September 2007.