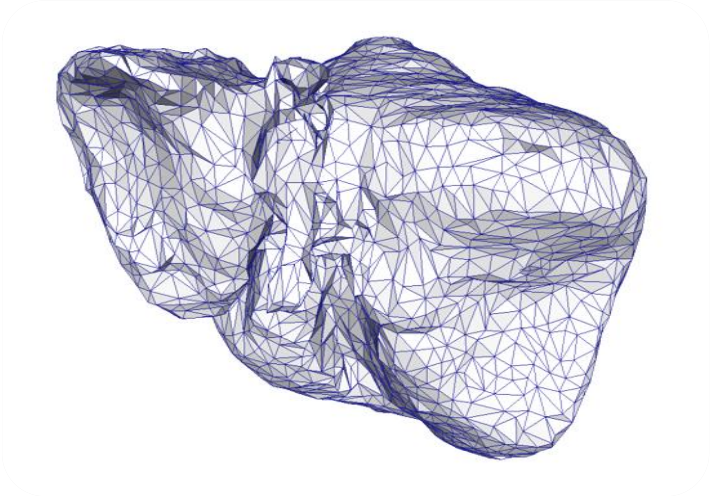
GRAZ UNIVERSITY OF TECHNOLOGY

INSTITUTE FOR COMPUTER GRAPHICS AND VISION

ITK Mesh Simplification using Error Quadrics and Directed Edges

Bachelor´s Thesis



Markus Pröll

proell@student.tugraz.at

Graz, September 2008

Supervision:

Univ. Prof. DI Dr. techn. Dieter Schmalstieg

Dipl.-Phys. Dipl.-Inform. Judith Mühl

KEYWORDS

level of detail, mesh simplification, error quadrics, directed edge data structure, vertex sets, border stitching

ABSTRACT

In 3D graphics surface models of objects are mostly represented by polygonal meshes. The ability to switch between several levels of details is essential to provide an appropriately sized input for all kinds of further processing steps. The simplification of those polygonal meshes can be carried out by already existing applications, which, however, were not created for medical imaging purposes. Most of them cannot be used for reducing meshes of medical 3D models. In this thesis we are pointing out the features necessary for dealing with meshes in the context of medical imaging.

Our main goal is to find a combination of simplification components and control mechanisms, which ensure that the topology, as well as the geometry, is preserved in a reduced version of a 3D model. The combination of *error quadrics* with a *directed edge data structure* defines the basis for our simplification tool. In order to keep the triangle meshes manifold, we have applied some completely new techniques for half edge meshes, *vertex sets*, *border stitching* and a solution for merging borders.

Acknowledgements

I would like to express my thanks to Univ. Prof. DI Dr. techn. Dieter Schmalstieg for making this project possible and having the trust in me to be up to this task. Nonetheless I would also like to thank my supervisor Dipl.-Phys. Dipl.-Inform. Judith Mühl for giving me great support throughout this project and sharing a lot of her great experience with me.

I also want to mention Christian Bauer. He provided the test data we could use during the development process and he also showed great endurance in defining the requirements for our application in detail.

Last but not least I want to thank Dipl.-Inform. Dr.-Ing. Sven Havemann and Dr. Markus Grabner who gave me a great understanding of this field of application in their courses. This was very motivating for me and it was a great experience exploring these topics in more detail.

Pledge

I hereby certify that the work presented in this bachelor's thesis is my own and that work performed by others is appropriately cited.

CONTENTS

1. INTRODUCTION

Polygonal meshes are commonly used to represent 3D models in computer graphics. They are available in different kinds of formats, levels of detail and can be found in various fields of applications, like computer games, 3D arts and CAD applications. In this work our focus will be on triangle meshes in the context of medical imaging and processing.

A common way to extract surface models of animal or human organs, bones and blood vessels is to generate a volumetric representation with either a CT[1] scan or different MRI[2] methods. Surface models extracted from this kind of data are typically represented by meshes which are very dense and consist of a high amount of homogenous triangles. These models contain every detail, which could be segmented and extracted from the original volumetric data, but even though they include many unnecessary triangles. The plain size of these models can easily be a bottleneck for further processing steps.

An appropriate model size can be achieved by simply reducing the amount of triangles in respect of preserving the geometry as well as the topology as much as possible. *Polygonal mesh simplification* (also referred to as *decimation*) is well covered by the topic *level of detail* (LOD). In combination with the field of medical imaging it is very essential not to lose any topological or geometric detail during all kinds of processing steps. A tumor for example, which is accidently "cured" during the simplification process will not be appreciated, so the preservation of details is mandatory and one of the main concerns in this project.

[1] CT: Computed Tomography

[2] MRI: Magnetic Resonance Imaging

1.1. Motivation

In Figure 1.1 we can see the polygonal mesh of a human liver. With 638,788 triangles it is a rather small model, but if we take a close look, as shown in Figure 1.2, we can see a very typical effect. The model consists of highly tessellated, plane surfaces, rather than single triangles. All triangles are nearly the same size and many of them do not influence the geometry of the model in any way. Edges and borders are defined by a very small amount of triangles.

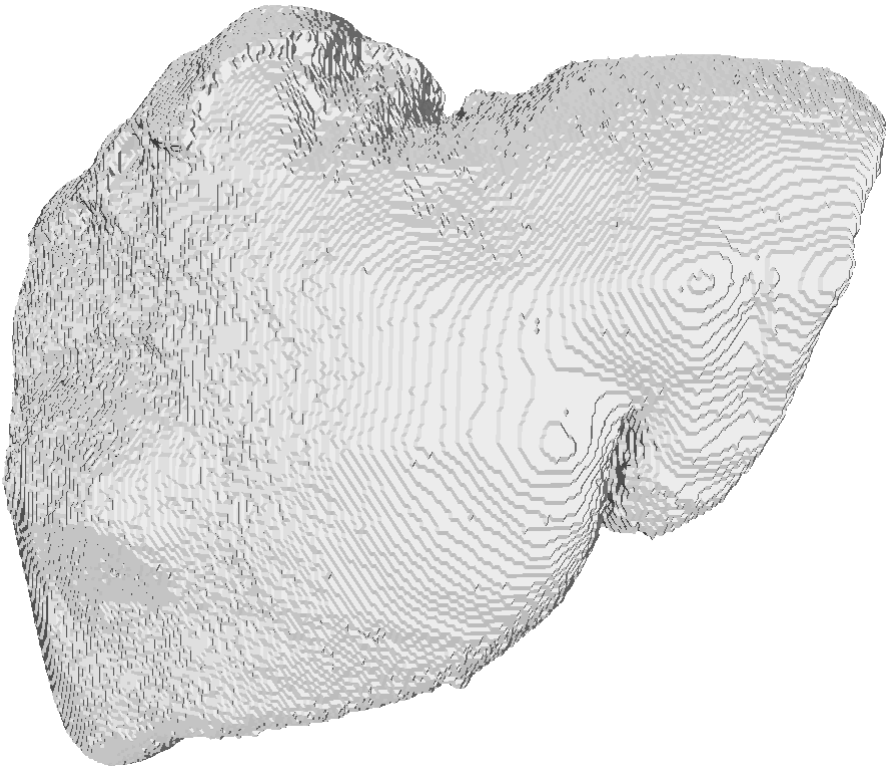


Figure 1.1: A polygonal mesh of a human liver (639k triangles).

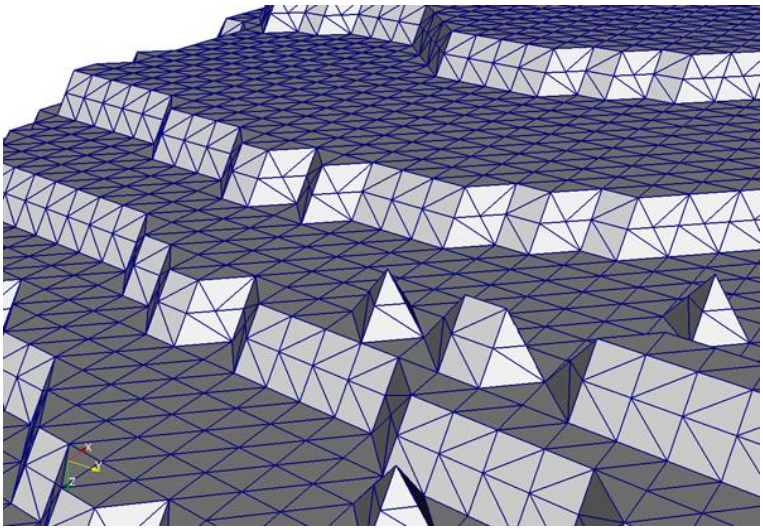


Figure 1.2: Zoomed view onto the liver mesh as seen in Figure 1.1.

This project was initiated by the need for a software application, embedded in the open-source software system *Insight Toolkit*[3] (ITK), which is capable of reducing the amount of triangles in such medical meshes until a certain user defined polygon-count or geometric error is reached. Of course there are already several applications, which are more or less able to perform this kind of task, but the translation between the frameworks native file format and those known by the simplification software packages was inconvenient and time consuming.

For reducing the meshes, which were generated by a *marching cubes*[4] algorithm, the simplification software *QSlim*[5] was used to generate lightweight models. Since the ITK toolkit did not provide any possibilities of reducing polygonal meshes it was necessary to use an external tool. The workflow inside the ITK framework was interrupted by exporting these

[3] Insight Toolkit: ITK is an open-source software system which provides algorithms for segmentation and registration tasks. It is written in C++ and is cross-platform [Kit08]. http://www.itk.org/index.htm

[4] Marching Cubes: This is an algorithm which extracts polygonal isosurfaces from volume representations. It is probably the standard method to perform this task [Lue03].

[5] QSlim: A software package developed by Michael Garland. It is based on the quadric-based simplification algorithm as described in [Gar99]. http://graphics.cs.uiuc.edu/~garland/software/qslim.html

models with several shell-scripts into the *QSlim* native *.smf*-file format. If *QSlim* was able to generate a satisfying simplification this translation process had to be reversed to perform further processing steps based on the reduced model within ITK. So one of the main goals of this project was to handle the simplification task with an ITK internal *MeshToMeshFilter*-class, which can easily be integrated into every kind of ITK application.

1.2. Overview

The following Chapter 2 deals with the related work our project is based on. An overview of the concepts and solutions we used in our tool is given and also some additional references are given to sources which are very close to the topics of this paper.

In Chapter 3 we point out the essentials of a simplification tool based on an already existing software package. With these prerequisites in mind we establish the requirements for our simplification tool in detail.

Chapter 4 covers the theory for this work. Starting from a general model of a simplification algorithm we discuss each component and create a selection which forms our simplification tool. We then take a look the algorithm sequence and show the steps a simplification process has to go through, for generating a reduced model.

The subsequent Chapter 5 deals with the challenges we were faced with when implementing a controlled simplification. A major part of this chapter deals with borders, since they do generate some pitfalls, which must be avoided during the simplification.

Chapter 6 and 7 cover unresolved challenges and the experiences gained from our project work. The final Chapter 8 deals with the installation procedure of our tool, followed by a summary of all possible program parameters which can be set to achieve the desired results.

2. Related Work

This thesis is fundamentally based on three pillars: *quadrics*, *LOD* and *directed edges*.

A great overview of the field of LOD is given in Luebke et al., *Level of Detail for 3D Graphics* [Lue03]. The authors present a majority of the most important simplification methods and describe in detail the pros and cons these algorithms provide. From this work we derived the general model of a simplification algorithm, which we will present in Chapter 4.2.

The work of Micheal Garland & Paul Heckbert, *quadric based simplification algorithm* [Gar97], provided many existing solutions to the challenges of this project. The error metric, the autors developed and integrated in their simplification tool *QSlim*, *error quadrics*, is used in our application to measure the error of a local simplification operation and defining a boundary for the maximum simplification error of an approximation (Chapter 4.2.3).

The third yet not less support was given Sven Campagna et al. [Cam98], with his definition of a completely new representation of triangle meshes. The so called *directed edge data structure* defines the state of the art for triangle mesh representations. This work was very well propagated by Sven Havemann who also provided the practical implementation of the data structure we use in his course script for *Modelling & Shape Description* [Hav06].

A comparison of the already existing simplification applications, with regard to quality of the approximation, is given in the report of Vitaly Surazhsky & Craig Gotsman, *A Qualitative Comparison of Some Mesh Simplification Software Packages* [Sur05]. The authors perform several simplification tasks with nine different commercial and open source tools to measure the quality of the resulting approximations in various ways.

3. Mesh Simplification Approaches

In this chapter we firstly explain some essential criteria for mesh simplification in the context of medical imaging and then provide an overview of all major requirements for our application.

3.1. Simplification Essentials for Medical Imaging

For the reason of a better explanation we point out the essentials in medical imaging in respect of the already mentioned simplification tool *QSlim*. This application was not explicitly designed for reducing medical meshes, but what is of importance when dealing with medical meshes? Which features make *QSlim* not applicable for this field of application? To answer these questions we take a closer look at *QSlim*.

QSlim is a very sophisticated tool, which produces approximations of triangle meshes in respect of speed and quality. This software package was developed by Michael Garland to represent his *quadric-based simplification algorithm* [Gar99]. The first version 1.0 was released in October 1997 and extended in version 2.1, which was last updated in July 2004. *QSlim* was not developed in the context of medical imaging purposes. One of its major design goals is that the topology of the original mesh can be altered during the simplification process.

"*I have also assumed that approximations need not maintain the topology of the original surface. Clearly, there are applications in which this is not true. In medical imaging, for instance, topology may provide critical information; depending on the application, preserving a hole in the heart wall may be much more important than preserving the exact shape of the surface.*", [Gar99].

The so called "*hole in the heart wall*" is a serious threat in medicine, better known as *ventricular septal defect* (VSD). Normally, the left and right ventricles of the human heart are divided by a wall. If this wall is not fully developed during the embryonic phase of a child, there can be one or more holes. Children which are born with this defect are also referred to

as so called "*blue-babies*". VSD affects about 1% of all new born children and can be treated with surgery [Chu06].

The main reason why the algorithm used in *QSlim* is able to alter the topology of a shape is the local simplification operator it uses; *vertex-pair contraction*. This operator takes any two vertices (v_i, v_j), which are *not* necessarily connected by an edge, of a triangle mesh M and contracts those into a vertex at position $\bar{v}$. [Gar99] denotes this operation as $(v_i, v_j) \rightarrow \bar{v}$. In Figure 3.1 we can see possible consequences of such a non-edge pair contraction. This operator is able to work on non-manifold triangle meshes and it is the reason why two completely separate components in a model can be merged in an approximation M'. Since the operator can choose any two vertices in a mesh the possible amount of contraction candidates would be $O(n^2)$ [Gra08]. For limiting this amount, a parameter τ can be set, so that all vertex pairs (v_i, v_j), which are not already connected by an edge, have to be within this distance τ. The set of all contraction candidates is then formed by the vertices (v_i, v_j), which are connected by an edge and all vertices $\|v_i, v_j\| < \tau$.

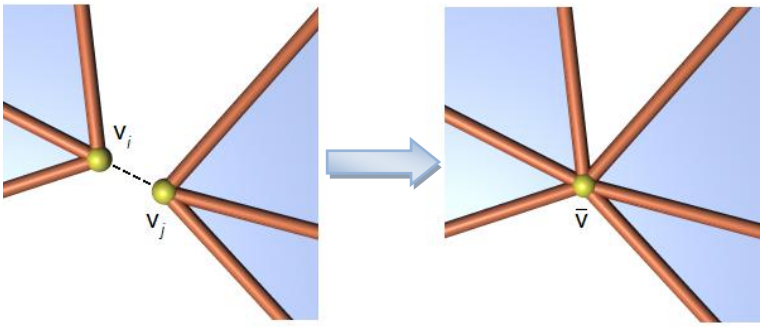


Figure 3.1: Non-edge pair contraction: a vertex-pair (v_i, v_j) which is not connected by an edge is contracted in the new position $\bar{v}$. Two separate components are connected in this operation.

Of course it is possible to choose the value zero for the maximum virtual edge length τ, but even in this case an approximation, which is not applicable for medical imaging purposes, can be the result. When two separate components in the model have one or more vertices placed in the same position in 3D space, caused by either insufficient accuracy of measurement or 32bit floats, the *vertex-pair contraction* operator will identify those as contraction candidates, which can lead to bad results. Maintaining the topology of the original object throughout the simplification is very important in medical imaging.

Apart from the topology of a mesh there are other properties we want to keep unchanged in a possible approximation, e.g. the total number of components. In case we have a surface model of a capillary blood vessel tree of the liver, we cannot expect that it has been segmented and extracted without any loss of connections in between. The result will be many unconnected components and we want to keep this amount during the simplification process. Even the smallest component should be represented as a volumetric entity in the final approximation. For example in *finite element methods*[6] (FEM) these set of small components can still have an influence and should not be lost.

A third major issue is the maintenance of manifoldness. *QSlim* is not aware of manifoldness, so it can produce results, which are not desirable in further processing. The set of non-manifold meshes includes all manifold meshes and can respectively be considered as more general, but for some applications it is necessary to have a completely manifold mesh. The transformation of a non-manifold mesh back into a manifold state can be very difficult and sometimes even impossible. So if the input can be considered as manifold in the first place the possibilities for further processing will not be limited, if the approximation is also a manifold polygon mesh.

[6] finite element method: "*The finite element method is a numerical analysis technique for obtaining approximate solutions to a wide variety of engineering problems.*", [Hue01].

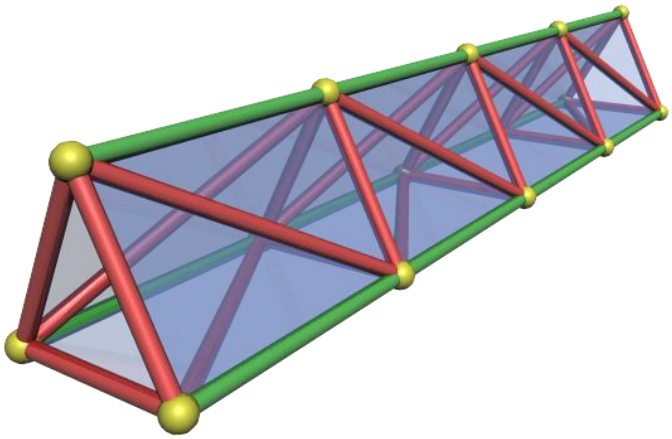


Figure 3.2: A simple tube made of triangles. All green edges can be collapsed without "closing" the tube. All edges marked in red will lead to a non-manifold object if they are contracted.

Problems which can occur, when manifoldness is no criteria can be best explained with an example. Let us think of a simple tube built of triangles, as shown in Figure 3.2. A simplification tool which does not keep meshes manifold is also allowed to contract for example edges around the diameter of this tube, which means that there will be no volume anymore. In case this tube represents a blood vessel, we erroneously alter a detail which can have a significant meaning for e.g. making a diagnosis or performing any simulations of blood circulation.

3.2. Project Requirements

We have now discussed some of the main issues a simplification application has to manage when dealing with medical meshes. In the following we sum up all the requirements, which need to be met by our simplification application.

As stated in the previous chapter, a mesh in detail can be described by several criteria. Manifoldness, borders, self-intersections, cells of zero area and connectedness, only to name a few. So we first have to determine which of these criteria are satisfied by our input

meshes extracted from volumetric data. Some of those can be regarded as very helpful and others are really hard to deal with. Table 3.1 shows the properties a mesh, which should be processed in our simplification tool, does not derogate from.

manifold	yes
borders	yes / no
triangles only	yes
cells of zero size	no
self intersections	yes / no

Table 3.1: Properties of meshes which are to be processed in our simplification tool.

An important requirement is that the simplification should be *fidelity-based*, which means that a certain user defined simplification error ε must not be exceeded. So when there are no more simplification steps possible, without breaking this boundary ε, the algorithm has to stop. It can also be useful to determine a minimum amount of polygons (*budget-based*) which should be reached during the simplification process, as long as the geometric error ε is in bounds.

The input meshes for our simplification tool are mostly available in two different file formats, namely: the *Open Inventor* file format (*.iv*) and the ITK file format (*.vtk*). They are stored as triangle meshes with a 32bit floating point precision. The 3D models represented by such meshes are not restricted to any certain class or type of objects. So it is important that the tool can e.g. deal with organs as well as complex blood vessel trees. A summary of all the requirements can be seen in Table 3.2.

Project Requirements
• ITK *MeshToMeshFilter*-class
• simplification of medical meshes (defined in Table 3.1)
• topology preservation
• maintain manifoldness
• generate a polygonal simplification with minimal geometric error
• stopping criterion based on simplification error and polygon count
• must operate on all different kinds of models

Table 3.2: Summary of all primary requirements for the simplification tool.

4. ITK Mesh Simplification

Now it is time for a first look behind the curtains. We give an insight into the design, features and possibilities our simplification tool provides. Later on, we discuss some challenges and pitfalls we had to manage during the development process and point out the solutions we provide.

4.1. Software and Framework

As already stated in the introduction, our main framework is the *Insight Toolkit* ITK. It is a cross-platform system and can be built with *CMake*[7] on Linux as well as on Microsoft Windows systems. By the time we started to program our application, the most recent version of ITK was 3.6, released in April 2008.

The programming was carried out with *Microsoft Visual Studio 2008* and was later on tested with the GCC 4.2.3 on a Linux system. Due to the fact that our application is highly generic, some of the templates we used were not compatible and had to be adjusted.

For the purpose of testing and presenting our results the *Visualization Toolkit*[8] (VTK) was used. VTK is very close to ITK and both can easily be connected in a processing pipeline. Since ITK is not aware of any visualization tasks, VTK takes care of presenting the results on screen. It was also of great help for debugging our application. In the final version we decided to separate this pipeline again, since an external viewer (e.g. *ParaView*[9]) provides plenty of options to view the results.

The import and export of the ITK internal mesh data structure to or from *.iv*, *.vtk*-files is handled by classes, which were already available and they could be used from the start.

[7] CMake: An open-source cross-platform build system for controlling the compiling process of software. http://www.cmake.org/

[8] Visualization ToolKit: VTK is an open-source C++ class library to perform visualization task in 3D computer graphics. http://www.vtk.org/

[9] ParaView: Open-source viewer for datasets of various formats. http://www.paraview.org/

4.2. Structure and Design

For creating a simplification application we mainly need four different components: a data structure, a local simplification operator, an error metric and a simplification framework (Figure 4.1). For each of these components there are several possibilities with different features to choose from. Below we discuss these main building blocks of our application and explain why we have selected the components we use.

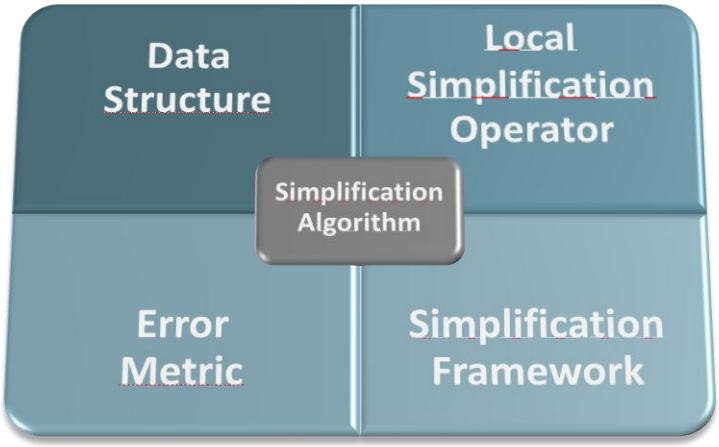


Figure 4.1: Main components of a simplification application.

4.2.1. Data Structure

The data structure, a simplification application is based on, is very essential in many ways. On the one hand it defines the efficiency in regard of speed of access and memory usage and on the other hand it specifies the type of mesh it is compatible with.

The mesh data structure offered by ITK was of course the first candidate number one, but after some testing we discovered that this highly generic structure cannot be dealt with easily. It is neither very efficient in memory usage, nor random access operations. A much better solution is the *directed edge data structure*, which fits perfectly our needs for dealing

with manifold triangle meshes. It was first introduced by Swen Campagna [Cam98] and in the opinion of Sven Havemann, "*it is probably the standard triangle mesh representation today*", [Hav06].

The implementation of the directed edge data structure we use in our project is templated with traits for vertices, edges and faces and so it can be regarded as highly generic. This kind of implementation is inspired by the work of Sven Havemann. A simplified sample of a tetrahedron and its data structure is shown in Figure 4.2.

Directed Edge Data Structure

index	vertex	mate
0	*v0*	11
1	*v1*	3
2	*v2*	8
3	*v2*	1
4	*v1*	10
5	*v3*	6
6	*v2*	5
7	*v3*	7
8	*v0*	2
9	*v0*	7
10	*v3*	4
11	*v1*	0

Vertices

index	x	y	z
v0	…	…	…
v1	…	…	…
v2	…	…	…
v3	…	…	…

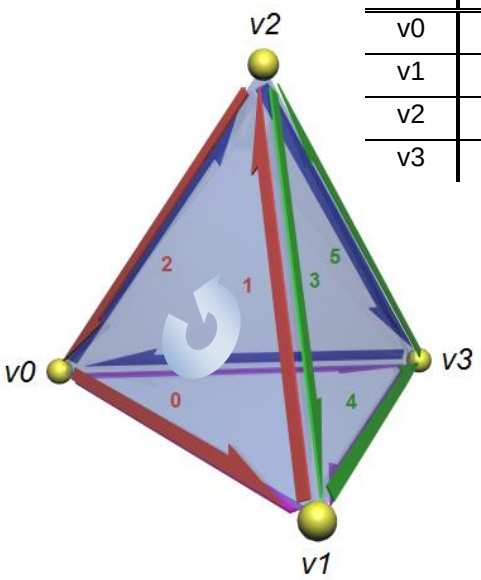


Figure 4.2: Sample *directed edge data structure* of a tetrahedron. In the table to the left we see the data structure, which contains the topology of the mesh. The vertex-table to the right contains the geometry of the mesh with the x, y and z coordinates of each vertex.

There is also another reason which makes the *directed edge data structure* really attractive; mesh traversal with iterators. Every half edge in the mesh can be considered as an iterator, which makes it possible to “crawl” over the mesh in a very convenient way. In general there are five operations which can be called on an iterator: *FaceCW()*[10], *FaceCCW*[11]*()*, *VertexCW()*, *VertexCCW()* and *EdgeMate()*. The operations are shown in detail in Figure 4.3. They make it possible to navigate over the mesh and retrieve relevant information about the neighborhood.

[10] CW: clockwise

[11] CCW: counterclockwise

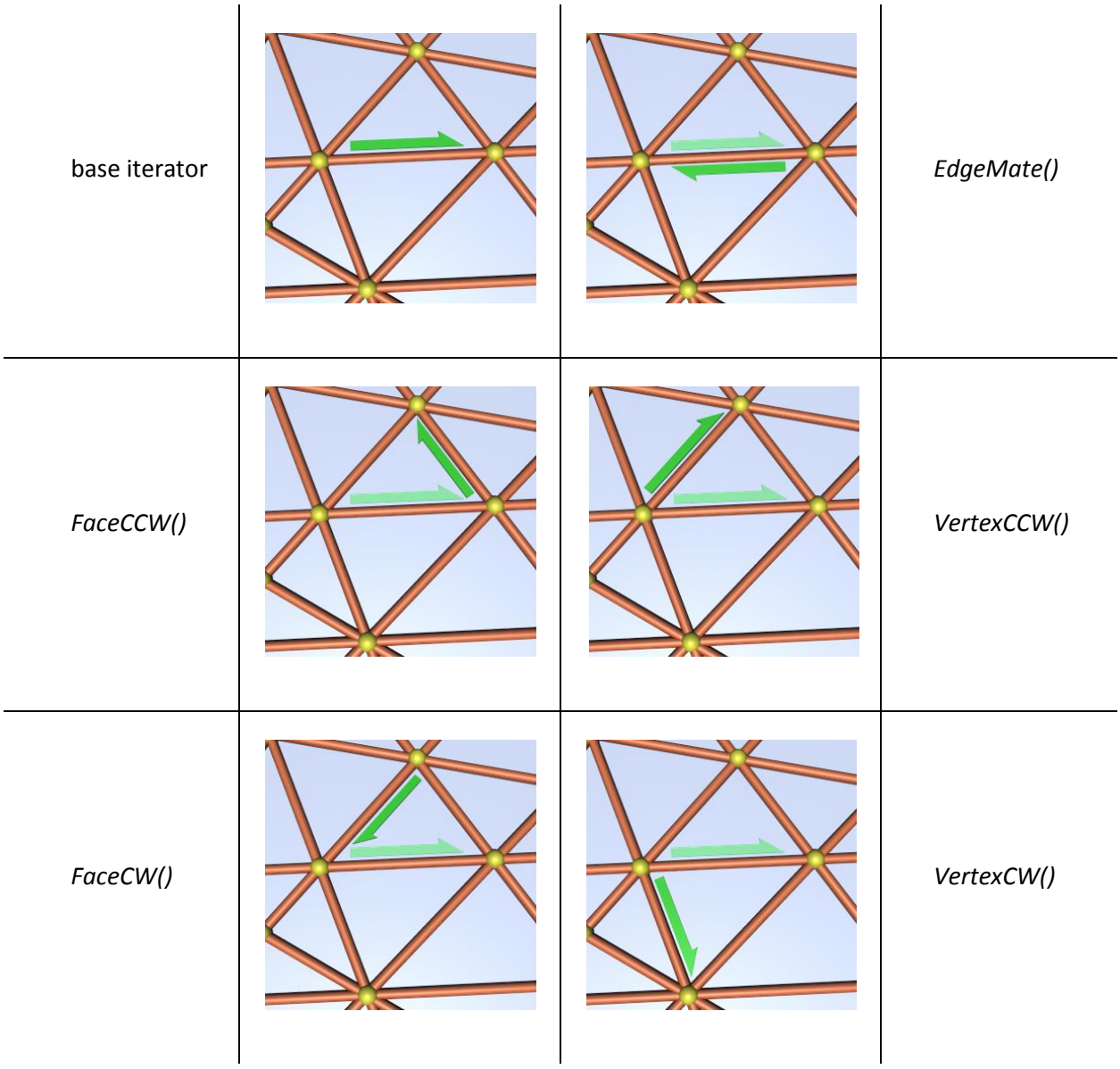


Figure 4.3: Overview of all operations which can be performed with a half edge iterator for mesh traversal.

4.2.2. Local Simplification Operator

We have already discussed the local simplification operator used by *QSlim* (see Chapter 3.1); *vertex pair contraction*. We pointed out that this operator can cause non-manifold results in an approximation. So what we need, is an operator which preserves manifoldness and works

in combination with our *directed edge data structure*. In our opinion a *full edge collapse* in combination with a foldover prevention is the best option, which meets those requirements.

A *full edge collapse* takes two vertices (v_a, v_b) which have to be connected by an edge and contracts those into a new vertex v_{new} [Lue03]. Contrary to the *half-edge collapse*, this operator keeps both old vertices (v_a, v_b), which we need later on to calculate the *mapping error* (see Chapter 4.3, *CalculateMappingError*) of the simplified mesh M'. An example of a *full edge collapse* is shown in Figure 4.4.

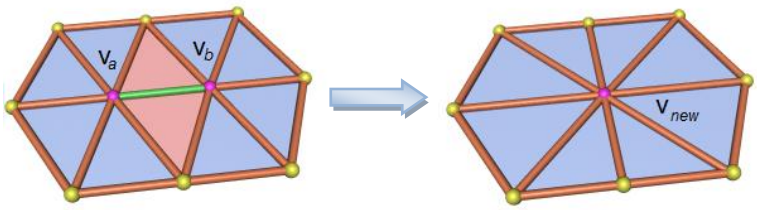


Figure 4.4: Full edge collapse: two vertices (v_a, v_b), which are connected by an edge, are collapsed to a new vertex v_{new}.

4.2.3. Error Metric

For the generation of a reduced approximation of a mesh we need to ensure that the derogation of the geometry is held as small as possible. For this reason we need an error metric which gives us the possibility of calculating some sort of costs for each contraction operation. Speed, fidelity and accuracy are the main qualities a good error metric has to provide. The so called *quadrics* meet all of those requirements. They were first introduced by Michael Garland and Paul Heckbert in 1997 [Gar97].

A *quadric* is a symmetric 4×4 matrix which can be stored within ten floats. They hold the information of a set of planes and have the attractive feature to be additive. Given a vertex v and a quadric Q the sum of squared distances between the vertex and the set of planes can be calculated by evaluating $(\mathrm{v}^{\mathrm{T}}\ \mathrm{Q}\ \mathrm{v})$ [Lue03]. For the purpose of a rough understanding of *quadrics* we provide a short description in the context of the simplification algorithm.

In the initial state of the simplification process each vertex v_i in the triangle mesh M is equipped with a *quadric* Q_i, which contains the planes v_i is surrounded by. Of course when we now evaluate the expression $(\mathrm{v}_i^{\mathrm{T}}\ \mathrm{Q}_i\ \mathrm{v}_i)$ there will be a squared distance sum of zero for all vertices of the Mesh M, because of the fact that v_i describes the intersection point of all those planes stored in Q_i. For each possible contraction candidate, formed by two vertices $(\mathrm{v}_i, \mathrm{v}_j)$ connected by an edge, their Quadrics are added, $\mathrm{Q}_i + \mathrm{Q}_j \rightarrow \mathrm{Q}_{new}$. What we can do now is to calculate an optimal position for our new vertex v_{new}, so that the expression $(\mathrm{v}_{new}^{\mathrm{T}}\ \mathrm{Q}_{new}\ \mathrm{v}_{new})$ evaluates to a minimum. This value describes the sum of squared distances between the new vertex v_{new} and all the planes which were part of Q_i, Q_j. We only have to evaluate the additions of quadrics and the multiplications of vertices with quadrics to measure the error as a squared sum of vertex-plain distances.

4.2.4. Simplification Framework

The simplification framework of a simplification algorithm is responsible for the *outer optimization* problem. With respect to some cost function (*quadrics*, see chapter 4.2.3) it selects the next contraction candidate from a queue; Contrary to the so called *inner optimization*, which performs the contraction operation as “cheap” as possible.

We decided to use *greedy queuing* [Lue03], because of the more accurate results it can produce. Of course there is some overhead we have to deal with, when the whole neighborhood is updated after an edge-contract. Since the runtime was not a major criterion in our application *greedy queuing* is the better way to go.

4.3. Simplification Algorithm Sequence

Now, that we have introduced the components for our simplification tool (see Figure 4.5), we explain how they do interact with each other. We will take a look at each sequential step (see Figure 4.6) in the algorithm and describe it in detail.

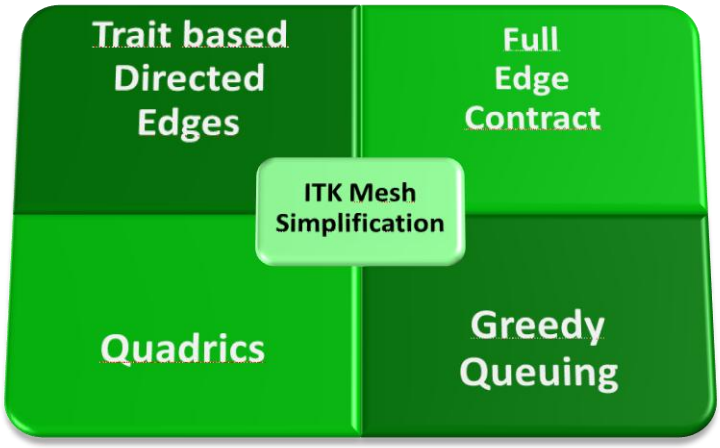


Figure 4.5: Overview of the components of the ITK mesh simplification tool.

1. ConvertITKToDEMesh

In this initial step the mesh is already read from file and needs to be converted into our *directed edge data structure*. The translation is done in linear time and all non-triangle cells in the ITK mesh will be ignored. An inconstant state of the mesh, which is about to be converted, can cause severe problems in this phase. For example if the mesh contains inexistent point indexes or wrong oriented faces (see Chapter 5.1) it cannot be converted properly. When the translation is complete, the ITK mesh is released to free its memory.

2. CollectInitialQuadrics

After filling our directed edge mesh, we start to calculate all the initial *quadrics* of each vertex. Since *quadrics* are additive (see Chapter 4.2.3) and initialized with zero values, it is possible to go through all the faces, calculate the quadric and add it to the corresponding vertices.

```
1   ConvertITKToDEMesh();
2   CollectInitialQuadrics();
3   ConstrainBorders();
    StitchBorders();
4   CollectContractionCandidates();
5   StartSimplification();
6   CalculateMappingError();
7   ConvertDEMeshToITK();
```

Figure 4.6: Sequential algorithm structure of the ITK Mesh Simplification tool.

3. ConstrainBorders & StitchBorders

These steps are necessary to deal with borders. They are explained in detail in Chapter 5.3.

4. CollectContractionCandidates

In this step of the algorithm we collect all potential contraction candidates, which means every *full-edge*[12] and each border edge has to be considered. Since the data structure is only

[12] full-edge: A full edge consists of two opposite half edges.

aware of half-edges, the corresponding edge-mates need to be flagged, to ensure not to insert every full-edge twice into the queue.

Each contraction candidate is then queued by its costs which have been calculated by evaluating the expression ($v_{new}^T Q_{new} v_{new}$) (see Chapter 4.2.3). The position of the vertex v_{new} can be determined in two different ways. If the user enables the optimal vertex positioning option, v_{new} is the position, which causes the lowest costs. Else, if this option is disabled or it is simply not possible to calculate a minimum, the position of one of the vertices which are part of the edge (v_i, v_j) is taken for the location of v_{new}. When this phase is finished the contraction queue is filled with all possible contractions.

5. StartSimplification

This is the main loop of the simplification process. The first contraction candidate in the queue is chosen for a possible contraction. If it is successful the neighborhood of the contraction candidate will be updated and reinserted into the queue (*greedy queuing*, see Chapter 4.2.4). In case the contraction could not be performed (this can have several reasons, which are discussed in Section 0) the costs for this contraction candidate are raised. The user can specify the number of maximum tries for each contraction candidate. If this number is exceeded this contraction candidate will be removed from the queue and never be reconsidered.

The loop is finished when:

a. there are no more contraction candidates left in the queue.
b. the amount of desired polygons is reached.
c. the defined error limit is reached.

6. CalculateMappingError

Since a *full edge contract* keeps all vertices from the original mesh M, it is possible to calculate the average and maximum *mapping error* by calculating the distance between the vertices of the original mesh M and the corresponding vertex in the approximation M'.

7. ConvertDEMeshToITK

The directed edge mesh of the final approximation M' is translated back to the output-mesh of the ITK *MeshToMeshFilter*-class.

5. Pitfalls and Solutions

In this section we will explore the challenges and pitfalls we discovered during the development process. Some of them could be solved with solutions already available. In the case of e.g. *border stitching* or *vertex sets*, we bring completely new answers to the field of directed edges and edge contraction.

5.1. Wrong Oriented Face

The orientation of faces is determined by their vertex-order. As usual a counterclockwise order denotes the front-face and a clockwise orientation the back-face of a polygon. In case one of the faces in a surface is flipped, so that the normal vector points in the opposite direction (shown in Figure 5.1), the *directed edge data structure* is not able to deal with this flipped face. For the reason of mesh traversal with iterators (see Chapter 4.2.1) it is necessary that each half-edge has an edge-mate oriented in the opposite direction.

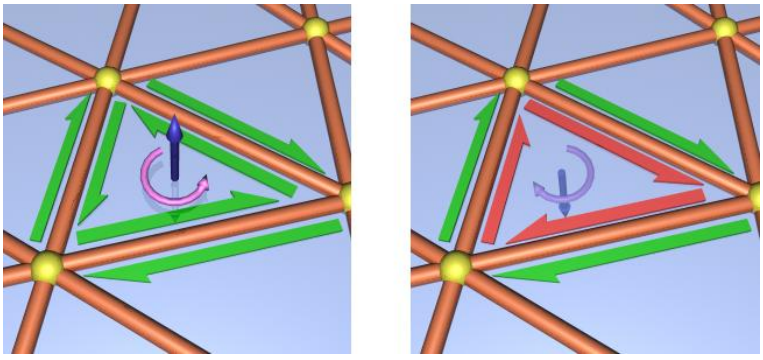


Figure 5.1: The face to the left is oriented in the right direction. Every half-edge has a mate which is oriented in the opposite direction. The right picture shows a wrong oriented face. Its half-edges have parallel mates.

Of course we could try to flip the face back again, but this would only work out in the optimal case, i.e. only a few single faces are turned around. What if by chance the first face

of the mesh we translate is the wrong oriented one and we flip all the others because they do not fit the first one?

By now have simply forbidden this kind of meshes in our application. A wrong oriented face will lead to an exception and the process of simplification is completely stopped.

5.2. Local Mesh Foldover

A local mesh foldover can happen during a *full edge contraction*. If the optimal position for the vertex v_{new} is placed outside a certain region, a foldover can be the result. Topologically the mesh is still consistent. To avoid this kind of problem we implemented a foldover prevention, which is described in [Gar99].

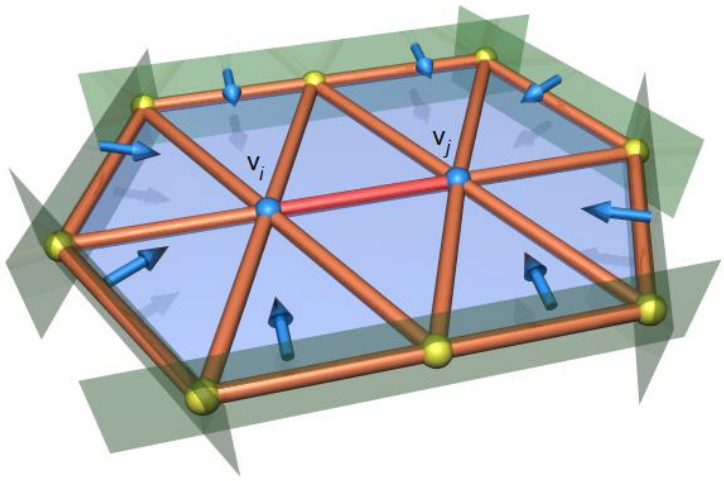


Figure 5.2: For preventing a local mesh foldover, the new vertex v_{new} has to be located within the region denoted by the blue arrows.

For each face around v_i a perpendicular plane is computed, which is located along the edges, which do not contain v_i. The same applies for v_j. This set of planes defines two regions and v_{new} has to be located in either one of those regions (see Figure 5.2).

5.3. Borders

Allowing borders in a mesh is a great relaxation of the manifold criterion. Borders cause a lot of exceptions. Nearly every operation has to deal with borders in a different way, but we provide a solution, how these exceptions can be reduced to a minimum. Later on we will describe this issue in detail, but let us first take a look at two other concerns evoked by borders.

5.3.1. Eating Borders

"*Eating borders*" are caused by the error metric we use; *quadrics* (see Chapter 4.2.3). If we do not treat borders in a special way, the algorithm will immediately start to perform the local simplification operator on edges which share a border-vertex. The reason for this effect is, that each quadric evaluates the costs of a possible contraction with respect to the set of planes, it is surrounded by. In border regions, the number of planes stored in the corresponding Quadric Q is much smaller than in regions, which are not close to a border (see Figure 5.3).

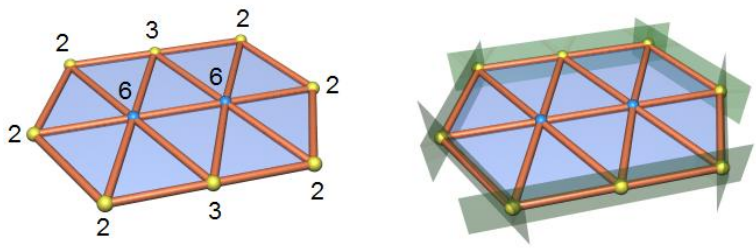


Figure 5.3: In the mesh to the left, the numbers denote the amount of surfaces in each quadric. For weighting the borders, their perpendicular planes are multiplied by a weighting factor and added up to the border-quadrics.

[Gar97] provides the solution of *border-weighting* for this effect. In short, each quadric, located at a border of the mesh, is summed up with a perpendicular plane for each edge the corresponding vertex is part of (see Figure 5.3). In addition those planes are weighted with a scalar factor, so that it is very unlikely that costs evaluated by those border-quadrics will be amongst the lowest.

5.3.2. Border Stitching

Borders in a triangle mesh cause a lot of exceptions. To provide an example, for *vertex sets* (see Chapter 5.4) it is essential to rotate around a vertex with an iterator operation like *VertexCCW* (Chapter 4.2.1). This rotation is finished, when the half-edge we started from is reached again. In case we run into a border edge, we would be stuck.

The most sensitive operation to borders is the local simplification operator. There are a lot of scenarios where an edge contract has to be aware of borders and treat them correctly, else it would cause an inconsistent state of our *directed edge data structure*. All in all we thought of a solution which delivers us from this massive exception handling, *border stitching*.

The expression itself explains already most of its functionality. What *border stitching* does is to add a topological border made of triangles to all of the already existing borders (see Figure 5.4). This might seem, like running in a hamster's wheel, but it is not. This process is only done once and all the additional half-edges we add up to our mesh are never considered as contraction candidates, since they have been collected in a previous state of the algorithm (see Figure 4.6). All operations which we perform will never get in contact with any border edges, since they will only run over the stitched border-area.

The additional faces which are added to our mesh are flagged and will not be taken into account, when the directed edge mesh is translated back into an ITK mesh again. This measure ensures that the mesh is not altered by *border stitching*.

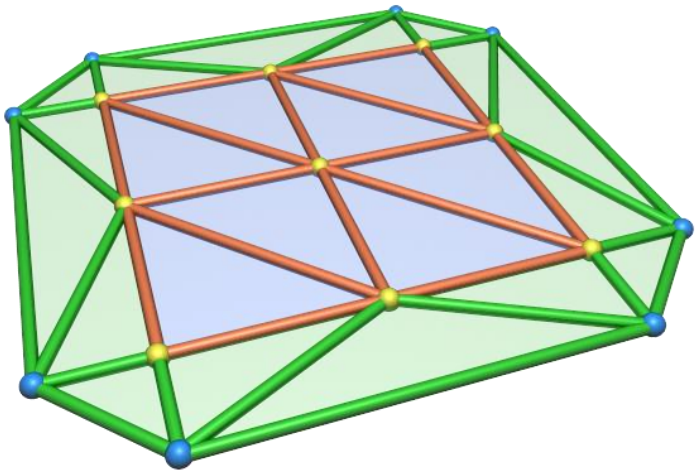


Figure 5.4: Border stitching adds a topological border (blue vertices) to the already existing borders (yellow vertices) to avoid exceptions in all operations.

5.3.3. Merging Borders

When we think of an object which has two or more borders, we need to ensure that none of those borders are merged during the simplification, because we want the genus of the mesh to be preserved. This effect can arise for example when two holes are very close to each other, only separated by at least one layer of triangles (see Figure 5.5). Any edge which connects two different borders in a mesh must not be contracted. Edges which are located along a border may be contracted, as long as the whole is at least made of three edges (triangle shaped).

To achieve this requirement, we assign a unique id to each vertex located at a border edge. This is done, when the mesh is fully imported into the *directed edge data structure*. We search all edges from the beginning to the end for any border edges. If one can be found we travel around the border with half edge iterators and mark the border vertices with an id.

In the local simplification operator we need to check if the contraction candidate is made of border vertices. If only one of the vertices v_i, v_j is marked with a border id the new vertex

v_{new} inherits this id. Otherwise if both vertices v_i, v_j are marked with an id, we need to ensure that they both share the same id. If they do so, the new vertex v_{new} is also tagged with the same id. In case v_i, v_j do not share the same id, the contraction must not be performed.

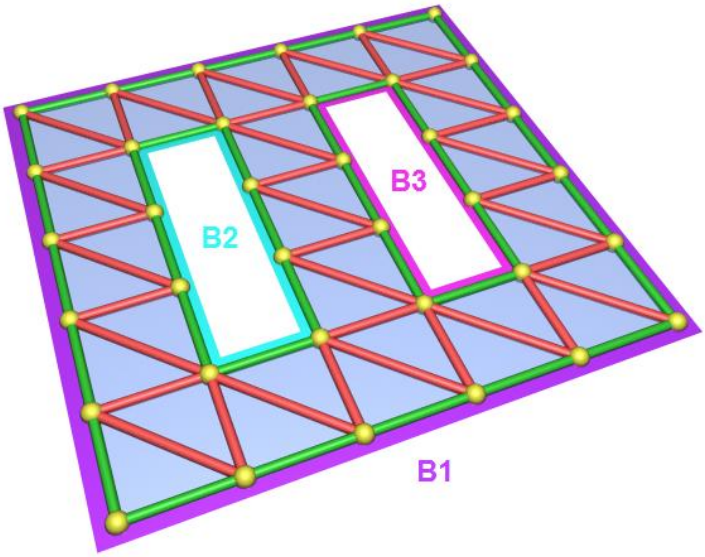


Figure 5.5: The edges in green may be contracted. All red edges connect two different borders and a contraction would change the genus of the mesh.

5.4. Vertex Sets

The technique of *vertex sets* is responsible for avoiding edge contractions which cause a non-manifold state of the mesh and additionally it preserves the number of components and holes in a mesh. We will first give some examples, where vertex sets ensure a controlled simplification.

We already pointed out (Chapter 3.1) that every component in a mesh should at least be represented by the smallest possible volumetric entity possible; the tetrahedron. So what

we need to ensure is that our application does not reduce this smallest platonic solid any further.

Considering the example of the simple tube made of triangles (see Figure 3.2), it is necessary not to collapse any edges around the diameter of the tube. In general we need to ensure that no contractions are possible, which cause a non-manifold approximation.

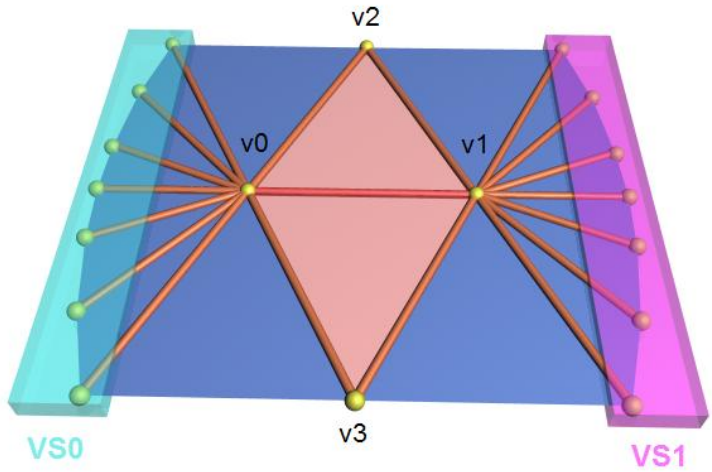


Figure 5.6: Vertex sets VS0 and VS1 for a contraction candidate v_0, v_1.

The preservation of holes is one of the major requirements to generate an optimal approximation for medical imaging purposes. As we already described in Chapter 5.3.3 we can prevent borders from merging, but we also need to ensure that holes are not closed. The smallest possible hole consists of at least three border edges. If one of the edges around this triangle shaped hole is contracted, the hole would be closed. This is what we want to pre-empt.

Each of those issues described above can be solved with the application of *vertex sets*. To explain this technique we will now take a closer look at the local surroundings of an edge

contraction. Figure 5.6 shows the region which is influenced during an edge contract. A vertex set for a vertex v_i is formed by all vertices which have a distance of one (if we think of a directed edges mesh as a directed graph) without the vertices which are part of the two faces which are about to be deleted.

Once we collected our *vertex sets*, there are three conditions, which have to be fulfilled:

(1) $VS0 \neq \emptyset$ and $VS1 \neq \emptyset$
(2) $VS0 \cap VS1 = \emptyset$
(3) $VS2 \cap VS3 = \emptyset$

If one of these 3 conditions does not apply the contraction is not performed.

6. FUTURE WORK

Our simplification application in general provides the basic functionality which is needed to simplify meshes. There are of course several aspects which are not yet completely solved. We will now discuss some of these open challenges and the missing features which are not integrated.

The condition of the input mesh is very essential for the generation of an approximation. Phenomena like wrongly oriented faces, zero area faces or other non-manifold conditions are not accepted. To increase the tolerance it would be necessary to develop some mechanisms which guarantee that these non-manifold meshes can be processed.

In the context of manifoldness in addition to a local foldover prevention a global foldover prevention would be an improvement which is very desirable in medical imaging. In the current implementation we cannot guarantee that the mesh does not intersect itself in regions, which are very close to each other. For example if we think of the lobes of a liver, which are located very close to each other, a global foldover prevention could ensure that these flaps stay separated in a reduced version of the original model.

In the context of massive mesh simplification our tool is limited by the main memory of the system our tool is running on. Some sort of clustering would eliminate this memory bottleneck. A technique to load and simplify only partial regions of the mesh would require a solution for correctly selecting those regions and ensuring a seamless approximation.

In regard of runtime improvements multithreading would be a good way to go. Since multiprocessor systems define the state of the art, a multithreaded simplification would definitely reduce the runtime by a significant amount. In combination with the previous mentioned clustering, this approach would represent an amazing improvement for the efficiency of the simplification application.

7. Conclusion

With the implementation of our simplification tool we show that *error quadrics* can be very well applied to a *directed edge data structure*. The combination of components and control mechanisms we have chosen make our tool highly applicable for reducing meshes without altering their topology and manifoldness. *Error quadrics* ensure that also the geometric error is held small and that approximation stays as close as possible to the original input mesh.

During the development process it was very difficult to keep the directed edge mesh consistent. This specific data structure is quite sensitive to inconsistencies, because of the mesh iterators, which are often used. If for example a *VertexCCW* rotation does not reach the start-iterator again this will lead to severe problems. In many cases the reason for this inconsistency is not discovered immediately. The results of a previous error are often shown in the subsequent phase of the simplification process. A lot of consistency checks are required and this is probably the greatest disadvantage of the *directed edge data structure*.

Our *border stitching* method defines a convenient solution for avoiding exceptions in all kinds of operations which are susceptible to borders. Any geometry can be chosen for those additional borders since it is only the topology which matters. The mesh itself is not altered by additional faces, as they are not included in the output mesh.

With *vertex sets* we introduced a technique to avoid several cases of non-manifold entities in a *directed edge data structure*. The sets can be collected very efficiently with mesh iterators and by sorting them with a heap, concurrencies can be found very fast. Since *vertex sets* are responsible for keeping the *directed edge data structure* consistent their application in our simplification tool is very essential.

8. Implementation Notes

In this section some practical advice how to install and use our simplification application is given.

8.1. Installation

The following is a step by step tutorial for setting up the ITK mesh simplification tool and running a first example. The requirements we assume here are a running Linux system with a *GCC*, *CMake* and *ITK* installed. For this tutorial we used the combination of *Ubuntu* 8.04, GCC 4.2.3, *CMake* 2.4, patch 7 and *ITK* 3.6. The configuration for the *ITK* build can be seen in Figure 8.1.

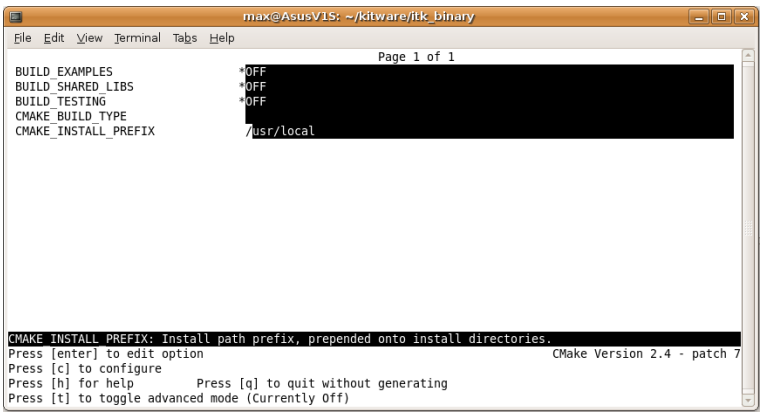


Figure 8.1: ITK building configuration for CMake.

(1) Call CMake for the mesh simplification tool. This can be done with:
`ccmake mesh_slim_linux_standalone`

(2) Now press [`c`] to configure,

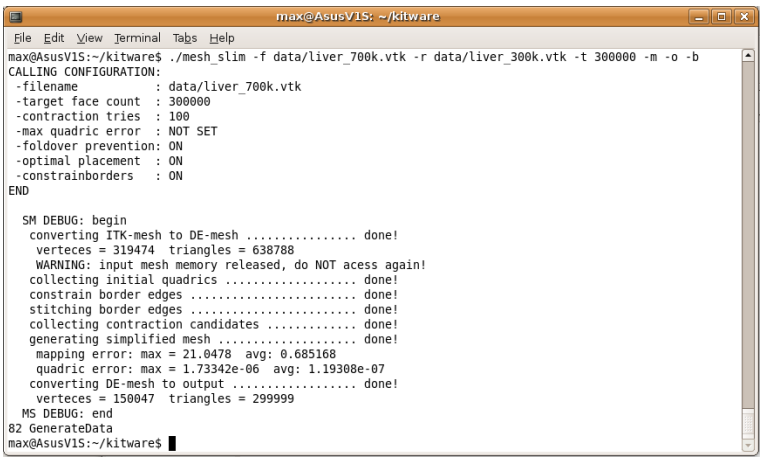
(3) followed by [`g`] to generate the makefile,

(4) compile the sources with calling `make`.

(5) Now that our application is built we can run a first test on a sample mesh (for a detailed description of the parameters read the following section):

```
./mesh_slim -f data/liver_700k.vtk -r data/liver_300k.vtk -t 300000
-m -o -b
```

(6) The tool is generating a simplified mesh of the sample with 300,000 (±1) triangles. The output can be seen in Figure 8.2.



```
max@AsusV1S: ~/kitware
File  Edit  View  Terminal  Tabs  Help
max@AsusV1S:~/kitware$ ./mesh_slim -f data/liver_700k.vtk -r data/liver_300k.vtk -t 300000 -m -o -b
CALLING CONFIGURATION:
 -filename           : data/liver_700k.vtk
 -target face count  : 300000
 -contraction tries  : 100
 -max quadric error  : NOT SET
 -foldover prevention: ON
 -optimal placement  : ON
 -constrainborders   : ON
END

  SM DEBUG: begin
   converting ITK-mesh to DE-mesh ............... done!
    verteces = 319474  triangles = 638788
    WARNING: input mesh memory released, do NOT acess again!
   collecting initial quadrics ................... done!
   constrain border edges ........................ done!
   stitching border edges ........................ done!
   collecting contraction candidates ............. done!
   generating simplified mesh .................... done!
    mapping error: max = 21.0478  avg: 0.685168
    quadric error: max = 1.73342e-06  avg: 1.19308e-07
   converting DE-mesh to output .................. done!
    verteces = 150047  triangles = 299999
  MS DEBUG: end
82 GenerateData
max@AsusV1S:~/kitware$
```

Figure 8.2: Output of the simplification tool during the reduction process.

(7) Finally, to view the results open *ParaView* and load the output file `liver_300k.vtk` (Figure 8.3).

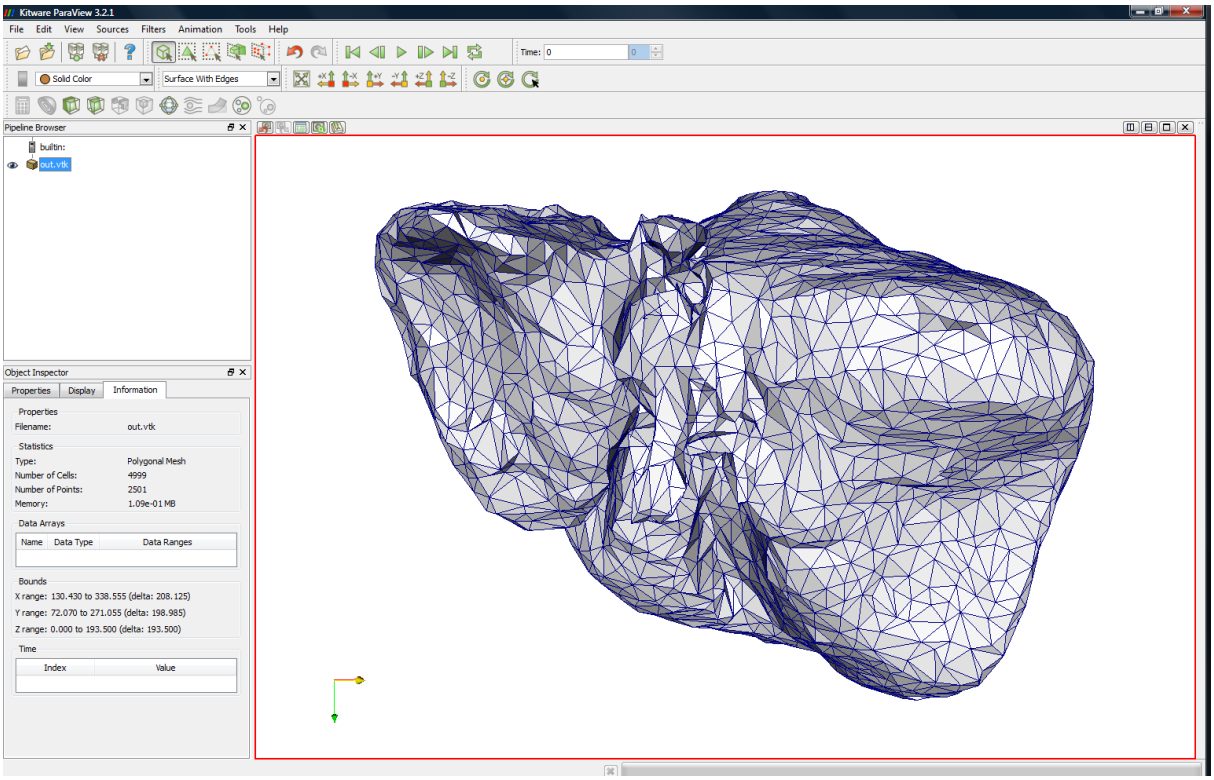


Figure 8.3: The results of a simplified mesh in ParaView.

8.2. Options & Parameters

For controlling the desired output approximation there are several parameters which can be set. There is a tradeoff if we want to increase the quality of the approximation, the runtime will also be raised.

`mesh_slim`

Invokes our simplification tool.

`-h`

Print a short help, where all parameters are listed and described.

`-f <filename>`

Set the filename for input mesh.

`-r <filename>`

Set the filename for output mesh.

`-t <uint>`

Set the target face count for the approximation. The simplification will stop when the given amount is reached and the simplification error is still in bounds. Also note that this amount can differ for the approximation by ±1 since an edge contract can only delete two faces.

`-c <uint>`

Set the maximum amount of tries for a contraction candidate. See Chapter 4.3, *StartSimplification* for a detailed description.

`-e <double>`

Set the maximum tolerated quadric error. See Chapter 4.2.3 for a detailed description.

`-m`

Enables the local mesh foldover prevention. See Chapter 5.2 for a detailed description.

`-o`

Enables optimal placement of target vertex. See Chapter 4.3, *CollectContractionCandidates*, for a detailed description. Note that enabling this option will increase the calculation effort significantly.

`-b`

Enables border constraining, this effect is described in Chapter 5.3.1.

Some samples for typical application calls:

```
./mesh_slim -f input_mesh.vtk -r output_mesh.vtk -e 0.001
./mesh_slim -f input_mesh.vtk -r output_mesh.vtk -e 0.001 -m -o -b
./mesh_slim -f input_mesh.vtk -r output_mesh.vtk -t 10000 -m -o -b
```

9. APPENDIX

9.1. BIBLIOGRAPHY

[Cam98] Campagna, Swen, Kobbelt, Leif and Seidel, Hans-Peter. 1998. *Directed Edges - A Scalable Representation for Triangle Meshes.* Erlangen, Germany : University of Erlangen, Computer Graphics Group, 1998.

[Chu06] Chun, Anne J. L. 2006. Medline Plus. [Online] 5 30, 2006. [Cited: 8 13, 2008.] http://www.nlm.nih.gov/medlineplus/ency/article/001099.htm.

[Gar97] Garland, Michael and Heckbert, Paul. 1997. *Surface Simplification Using Quadric Error Metrics.* s.l. : Proceeding of SIGGRAPH '97, 1997.

[Gar99] Garland, Michael. 1999. *Quadric-Based Polygonal Surface Simplification.* Pittsburgh : School of Computer Science, 1999.

[Gra08] Grabner, Markus. 2008. *EZG2 course slides: level of detail.* Graz : Institute for Computer Graphics and Vision, 2008.

[Hav06] Havemann, Sven. 2006. *Triangle Meshes.* Graz, Austria : Modelling and Shape Description, Course Script, 2006.

[Hue01] Huebner, Kenneth H., et al. 2001. *The Finite Element Method for Engineers, Fourth Edition.* s.l. : Wiley-IEEE, 2001. ISBN 0471370789, 9780471370789.

[Kit08] Kitware. 2008. NLM Insight Segmentation & Registration Toolkit. [Online] 6 4, 2008. [Cited: 8 14, 2008.] http://www.itk.org/index.htm.

[Lue03] Luebke, David, et al. 2003. *Level of Detail for 3D Graphics.* San Francisco : Morgan Kaufmann Publishers, 2003. 1-55860-838-9.

[Sur05] Surazhsky, Vitaly und Gotsman, Craig. 2005. *A Qualitative Comparison of Some Mesh Simplification Software Packages.* s.l. : CMA/IFI, University of Oslo, 2005.

9.2. List of Figures

9.3. List of Tables